ELAN
USER MANUAL

May 9, 1997
ELAN V2.00
BETA version

Peter Borovanský
Claude Kirchner
Hélène Kirchner
Pierre-Etienne Moreau
Marian Vittek

ELAN: **USER MANUAL**

Authors: Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Marian Vittek

INRIA Lorraine & CRIN
Campus scientifique
615, rue du Jardin Botanique
BP101
54602 Villers-lès-Nancy CEDEX
FRANCE

E-mail: Peter Borovansky@loria.fr, Kirchner@loria.fr, Pierre-Etienne.Moreau@loria.fr, Vittek@loria.fr

---

elan n. **1**. Enthusiastic vigor and liveness. **2**. Style; flair. [Fr < OFr. *eslan*, rush < eslancer, to throw out: es-, out (< Lat. ex-) + *lancer*, to throw (< LLat. *lanceare*, to throw a lance < Lat. *lancea*, lance).]

The American Heritage Dictionary

Contents

Foreword

This manual presents the version V2.00 of the ELAN language and of its environnement.

With respect to the last distributed version (V1.17), the new features are:

- several main changes in the syntax, in order to uniformize the language constructions and to add new features like new built in strategies, the use of patterns in where, rule visibility for rules and strategies labels, the automatic construction of selectors and modifiers.
- the availability of the compiler that allow ELAN programs to run at speed comparable to compiled functional languages,
- an extended strategy language giving to the user a much broader expressive power in the expression of strategies,
- an extended user interaction loop,
- and of course the correction of several bugs.

This is a preliminary state of the manual for ELAN version V2.00.
Your comments are welcome.

Chapter 1

A short description of ELAN

This chapter presents in a top-down approach the ELAN main features. If you are beginning to use ELAN, you should certainly have a look at this chapter first. On the contrary, Chapter 3 presents a bottom-up complete description of the language.

1.1 What could you do in ELAN?

Relying on the premiss that inferences rules can be quite conveniently described by rewrite rules, we started in the early nineties to design and implement a language in which inference systems can be represented in a natural way, and executed reasonably efficiently.

This leads us quickly to formalise such a language using the rewriting logic introduced by J. Meseguer [Mes92] and to see ELAN as a logical framework where the frame logic is rewriting logic extended with the fundamental notion of strategies [Vit94, KKV95]. We call a rewrite theory and an associated strategy a *computational system*.

Thus in ELAN, a logic can be expressed by specifying its syntax, its inference rules and a description of the way you would like to apply these rules. The syntax of the logic can be described using mixfix operators as in the OBJ language [GKK+87] or SDF [HHKR89]. The inference rules of the logic are described by conditional rewrite rules with `where` assignments allowing to introduce variables local to a rule.

From this description of the logic and a description of the signature considered, we infer automatically a computational system consisting of a rewriting theory plus strategies.

Since we wanted ELAN to be modular, with a syntactic level well-adapted to the user needs, the language is designed in such a way that three programming levels are possible.

- First the design of a logic is done by the so-called *super-user* with the help of a powerful preprocessor.
- Such a logic can be used by the (standard) *user* in order to write a specification.
- Finally, the *end-user* can evaluate queries valid in the specification, following the semantics described by the logic.

This corresponds to the general diagram given in Figure 1.1. The query is interactively given by the end-user at the ELAN prompt level.

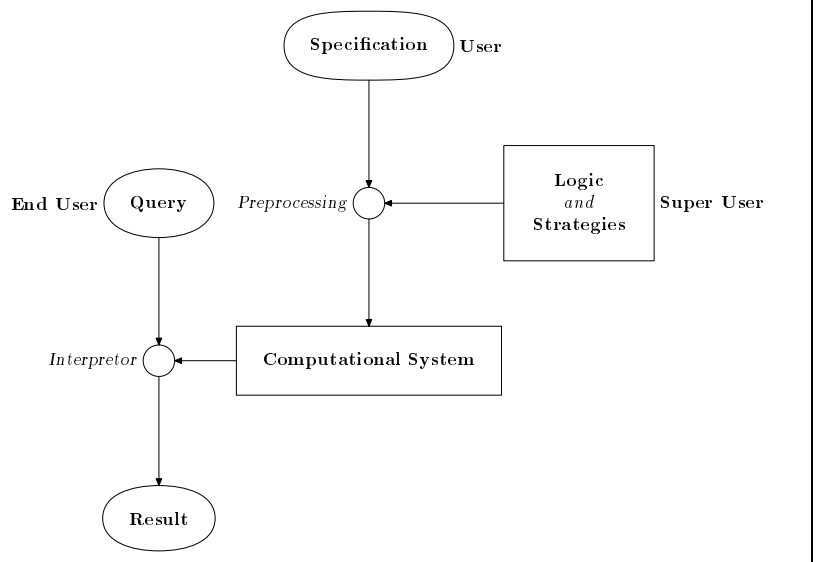


Figure 1.1: Elan: general organisation

1.2 A very simple example

Let us fully detail a complete example. The next module illustrates what the programmer has to write in order to describe the derivative operation on simple polynomials. The first part contains:

- the module name: `poly1`
- modules you want to import: `int` (the integer library module)
- sort declaration: `variable` and `poly`
- operators definitions: constructors and functions declarations

The second part contains the computation definition. Given a query, ELAN repeatedly normalizes the term using unlabelled rules. This is done in order to perform functional evaluation and thus it is recommended to the user to provide a confluent and terminating unlabelled rewrite system to ensure termination and unicity of the result. This normalisation process is built in the evaluation mechanism and consists in a leftmost innermost normalisation. This yield always a *single* result.

You can control the ELAN construction of terms in giving associativity or priority to operators.

```
module poly1                                                          1
import global int;                                                    2
end                                                                   3
sort variable poly;                                                   4
end                                                                   5
operators global                                                      6
 X                  :                    variable;                  7
 @                  :  ( variable )      poly;                      8
```

```
 @                  :  ( int )           poly;                                    9
 @ + @              :  ( poly poly )     poly assocRight pri 1;                   10
 @ * @              :  ( poly poly)      poly assocRight pri 2;                   11
 deriv(@)           :  ( poly)           poly;                                    12
end                                                                               13
                                                                                  14
rules for poly                                                                    15
          p1, p2 :  poly;                                                         16
          n :  int;                                                               17
local                                                                             18
 [] deriv(X)                  => 1                                  end           19
 [] deriv(n)                  => 0                                  end           20
 [] deriv(p1+p2)              => deriv(p1)+deriv(p2)                end           21
 [] deriv(p1*p2)              => deriv(p1)*p2 + p1*deriv(p2)        end           22
end                                                                               23
end                                                                               24
```

The top level of the logic description given in the following module describes the way to run the system. The logic declaration just introduces the sorts of interest (query and result) and defines `.eln` modules to be imported: this is the "top level".

```
LPL poly1 description                                                             1
          query of sort              poly                                         2
          result of sort             poly                                         3
          import                     poly1                                        4
          start with                 ()query                                      5
end                                                                               6
```

Once defined, we can use the two previous modules:

```
huggy ElanExamples % elan poly1.lgi
enter query term finished by the key word 'end':
  deriv(X) end

[] start with term: deriv(X)
[] result term:       1

enter query term finished by the key word 'end':
  deriv(3*X*X + 2*X + 7) end

[] start with term: deriv(3*X*X+2*X+7)
[] result term:       0*X*X+3*1*X+X*1+0*X+2*1+0
```

1.3 A more generic example

The next example, introduces more features from ELAN. It consists of the specification of elementary polynomials built over a finite set of variables, and the derivative functions to compute the derivated form with respect to a given variable. Tasks are divided as follows:

1. the super-user describes in a generic way the derivative and factorisation inferences, i.e. a logic for polynomial differentiation,

2. the user gives the specification of an algebra in which (s)he wants to derivate polynomials; in this case, this is quite simple since it amounts to specify the variables of the considered polynomial algebra,

3. the end-user gives a differentiation problem.

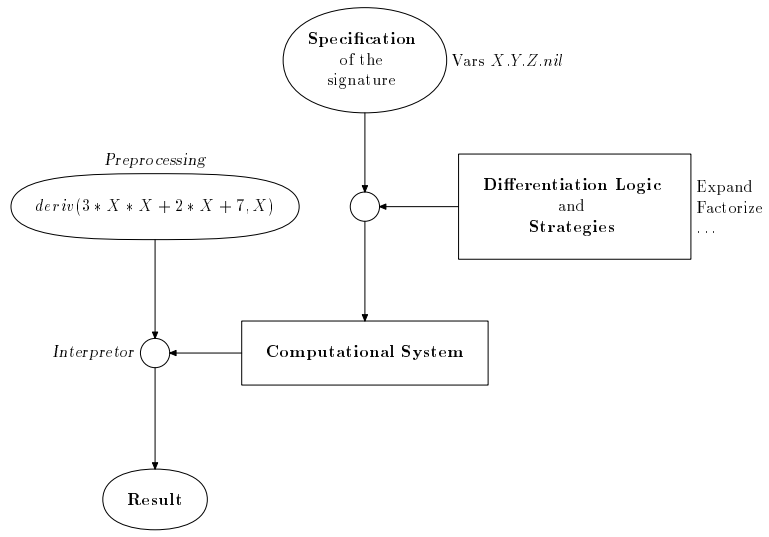


Figure 1.2: Elan: how to implement polynomial differentiation

The diagram of Figure 1.1 thus instantiates as described in Figure 1.2.

The description of the logic and of the specification are done in the ELAN syntax described in the Chapter 3 and it can be further extended by the super-user. These descriptions are assumed to be done in files with specific extensions:

1. `.lgi` for the top level logic description, this file is written by the ELAN super-user,
2. `.eln` for a module used in a logic description, this file is also written by the ELAN super-user,
3. `.spc` for a specification, i.e. a program written in the defined logic by an ELAN user.

The `.eln` module of our example is the following:

```
module poly2[Vars]                                                          1
import global int Vars eq[variable] identifier list[identifier];            2
end                                                                         3
sort variable poly;                                                         4
end                                                                         5
operators global                                                            6
 FOR EACH Id:identifier SUCH THAT Id:=(listExtract) elem(Vars) :            7
 { Id :  variable; }                                                        8
                                                                            9
 @                    :  ( variable ) poly;                                 10
 @                    :  ( int ) poly;                                      11
 @ + @                :  ( poly poly ) poly assocRight pri 1 (AC);          12
 @ * @                :  ( poly poly) poly assocRight pri 2 (AC);           13
 deriv(@,@)           :  ( poly variable) poly;                             14
end                                                                         15
                                                                            16
rules for poly                                                              17
          p, p1, p2 :  poly;                                                18
          x, y :  variable;                                                 19
          n :  int;                                                         20
local                                                                       21
```

```
 [] 0+p                    => p                                  end          22
 [] 0*p                    => 0                                  end          23
 [] 1*p                    => p                                  end          24
 [] deriv(x,x)             => 1                                  end          25
 [] deriv(y,x)             => 0 if x != y                        end          26
 [] deriv(n,x)             => 0                                  end          27
 [] deriv(p1+p2,x)         => deriv(p1,x)+deriv(p2,x)                   end   28
 [] deriv(p1*p2,x)         => deriv(p1,x)*p2 + p1*deriv(p2,x)           end   29
end                                                                           30
end                                                                           31
```

Then, the logic declaration describes the way to parse the specification file that contains a finite list of variables.

```
LPL poly2 description                                                         1
 specification description                                                    2
 part Vars          of sort list[identifier]                                  3
                    import identifier list[identifier]                        4
 end                                                                          5
                                                                              6
 query of sort                          poly                                  7
          result of sort                poly                                  8
 import                                 poly2[Vars]                           9
 start with                             () query                              10
end                                                                           11
```

To instanciate the *Logic* and build the *Computational System*, you have to give a specification. An example of such specification is:

```
specification someVariables                                                   1
 Vars               X.Y.Z.nil                                                 2
end                                                                           3
```

It contains a list of variables that will be read and transformed into a rewrite rule: `Vars => X.Y.Z.nil`

The `FOR EACH` preprocessor construction used in the `poly2.eln` module performs a textual replacement that extracts identifiers `X`, `Y`, `Z` from the list `X.Y.Z.nil` (this list built and managed by the parametization mechanism) and declare them of sort `variable`.

You can notice that operators `*` and `+` are now declared as Associative and Commutative, which is helpful to implement some simplification rules. One rule contains a conditionnal part.

Now let us call ELAN with the specification `someVariables.spc`:

```
huggy ElanExamples % elan poly2 someVariables.spc
enter query term finished by the key word 'end':
  deriv(3*X*X + 2*X + 7 , X) end

[] start with term: deriv(3*X*X+2*X+1,X)
[] result term:       X+X*3+2
```

The result has been simplified in a better way, but it is still difficult to read due to unparenthesis form.

1.4 An extended example

This last example shows you how to play with labelled rules and strategies.

Unlabelled rules are applied repeatedly, at any position, to normalize the term. A labelled rule is applied only if a strategy tries to apply it. In this case, the rule is applied on top position of the term. In the next module, you can recognize them: names are given between brackets.

You can define strategies in the same way you defined rules and use them in local affectation parts: `where`. When a rule is applied, before constructing the right-hand-side, conditional parts are evaluated and variables involved in local affectation parts are instanciated by applying a strategy on a term:

```
module poly3[Vars]                                                                  1
import global int Vars eq[variable] identifier list[identifier];                    2
end                                                                                 3
sort variable poly;                                                                 4
end                                                                                 5
operators global                                                                    6
 FOR EACH Id:identifier SUCH THAT Id:=(listExtract) elem(Vars) :                    7
 { Id :  variable; }                                                                8
 @                  :  ( variable ) poly;                                           9
 @                  :  ( int ) poly;                                                10
 @ + @              :  ( poly poly ) poly assocRight pri 1 (AC);                    11
 (@ + @)            :  ( poly poly ) poly alias @ + @:;                             12
 @ * @              :  ( poly poly) poly assocRight pri 2 (AC);                     13
 (@ * @)            :  ( poly poly) poly alias @ * @:;                              14
 deriv(@,@)         :  ( poly variable) poly;                                       15
end                                                                                 16
                                                                                    17
rules for poly                                                                      18
          P, p1, p2, p3, p4, p5          :  poly;                                   19
          e1, e2, e3, e4                 :  poly;                                   20
          x, y                           :  variable;                               21
          n, n1, n2, n3                  :  int;                                    22
global                                                                              23
 [] P+0                       => P                                    end           24
 [] P*0                       => 0                                    end           25
 [] P*1                       => P                                    end           26
 [] n1*n2                     => n where n:=() n1*n2                  end           27
 [] deriv(x,x)                => 1                                    end           28
 [] deriv(y,x)                => 0 if x != y                          end           29
 [] deriv(n,x)                => 0                                    end           30
 [] deriv(p1+p2,x)            => p3                                                 31
                    where p3:=(simplify) deriv(p1,x)+deriv(p2,x)                    32
 end                                                                                33
 [] deriv(p1*p2,x)            => p5                                                 34
                    where p4:=(simplify) p1*deriv(p2,x)                             35
                    where p3:=(simplify) deriv(p1,x)*p2                             36
                    where p5:=(simplify) p3+p4                           end        37
local                                                                               38
 [factorize] P + p1+p1                    => P + 2*p1                    end        39
                                                                                    40
 [factorize] P + n1*p1 + n2*p1 => P + n3*p1                                         41
                    where n3:=()n1+n2                                    end        42
                                                                                    43
 [factorize] P + (p1*p2) + (p1*p3) => P + p4                                        44
                    where p4:=(simplify) p1*(p2+p3)                      end        45
                                                                                    46
 [expand] P + (p1+p2)*(p3+p4) => P+e1+e2+e3+e4                                      47
                    where e4:=(simplify) p2*p4                                      48
                    where e3:=(simplify) p2*p3                                      49
                    where e2:=(simplify) p1*p4                                      50
                    where e1:=(simplify) p1*p3                           end        51
 [expand] P + (p1+p2)*p3 => P + e1+e2                                               52
                    where e2:=(simplify) p2*p3                                      53
                    where e1:=(simplify) p1*p3                           end        54
end                                                                                 55
                                                                                    56
strategies for poly                                                                 57
global                                                                              58
 [last_simplify]                                                                    59
          repeat*(dc(expand)) ; repeat*(dc(factorize))                              60
 end                                                                                61
 [simplify]                                                                         62
          repeat*(dc(expand)) ; repeat*(dc(factorize))                              63
```

```
      end                                                              64
    end                                                                65
    end                                                                66
```

And now we can use it:

```
huggy ElanExamples % elan poly3 someVariables.spc
enter query term finished by the key word 'end':
  deriv(3*X*X + 2*X + 7 , X) end

[] start with term: deriv(3*X*X+2*X+1,X)
[] result term:       ((X*6)+2)
[] end
```

Chapter 2

How to use ELAN

2.1 How to get and install ELAN

The latest version of the ELAN system includes the language interpretor, the compiler, the standard library and a lot of applications and examples. It runs on the following architectures: DEC-ALPHA, SUN4 and Intel-PC, and it can be obtained from the ftp server `ftp.loria.fr` from the directory `/pub/loria/protheo/softwares/Elan/dist.2.00.tar.gz`. This package contains a binary executable file of ELAN, the library of more than 80 basic ELAN modules, several basic ELAN applications and examples, the manual of ELAN in `.dvi` form, and the ELAN pretty-printing utility. This downloaded distribution file, after un-zipping and un-tarring, has the following structure:

```
bin/                - the executable files & ELAN tools
  alpha/              - executable image for the architecture DEC-ALPHA
  sun4*/              - executable images for the architectures SUN4 (sun4c, sun4m, sun4u)
  i86pc/              - executable image for PCs, under OS-Solaris
  i586/               - executable image for PCs >= i386, under OS-Linux
applications/       - a several ELAN applications
  FastCompletion/     - the fast completion algorithm
  Hou/                - the higher-order unification via explicit substitutions
  MinelaComp/         - the reflexive interpretor of a subset of ELAN
  Poly/               - the algebra of polynomials (several variations on derivations)
  Prolog/             - the vanilla interpretor of Prolog (SLD-resolution)
  Prover/             - a mechanical theorem prover (the calculus of sequent)
  Rewriting/          - the first-order rewriting
  Robot/              - several defined depth-first search strategies
  Simplest/           - classic: Fibonacci, 8-queens, primes numbers, etc.
  Strategies/         - examples of defined strategies
  Unification/        - the syntactic matching, unification, AC-matching (???)
contributions/      - potential contributions to ELAN
  Csp/                - Constraint satisfaction problem
elanlib/            - a library of ELAN modules
  Compiled/           - the architecture dependent part of the library
    alpha/ i86pc/ i586/ sun4*/      (for the compiler only)
  common/             - common files of quoted and non-quoted libraries
  quote/              - the quoted library (obsolete, only for the compatibility)
  noquote/            - the non-quoted part of the library
  strategy/           - the library of defined strategies
  help.txt            - the help file of the command line interpretor
manual/             - manual.dvi
```

If the distribution file has been uncompressed into a directory PWD, the installation of ELAN can be completed by:

- including the path $PWD/bin/`uname -m`$ into your environment variable $PATH$, and
- setting-up your environment variable $ELANLIB$ to $PWD/elanlib$.

If any problems occur during your installation, we recommend to read the `README` file at the top of the directory containing the ELAN distribution. The completely installed version of the system ELAN takes for one architecture about 12 MB of disk-space, and any additional architecture takes about 2MB.

In case you have any problems, questions or comments about ELAN, or just only to be keeping in touch with the ELAN team, please use the following e-mail address: `elan@loria.fr`. ELAN is issued from the PROTHEO research team, information about this research project can be found in the following url: `http://www.loria.fr/equipe/protheo.html/`.

2.2 How to run the ELAN interpretor

You can now execute ELAN by just typing the `elan` command with the following usage:

```
        ******** ELAN version 2.00 (17/04/97.09:58) ********

         (c)   INRIA-Lorraine  &  CRIN,   1994, 1995, 1996, 1997

elan usage : elan [options] lgi_file [spc_file]
    options :
        --dump, -d              : dump of signature, strategies and rules
        --trace, -t [num]       : tracing of execution (default max)
        --statistic, -s/-S      : statistics: short/long
        --warningsoff, -w       : suppress warning messages of the parser
        --quiet, -q             : quiet regime of execution
        --batch, -b             : batch regime (no messages at all)
        --elanlib               : elanlib
        --secondlib, -l lib     : second elan library (by default ..)
        --command, -C           : command language
        --compiler, -c          : use compiler
        --optimise, -O          : optimise compiled code
        --deterministic, -n     : use deterministic library (for compiler only)
        --exe                   : compile into an independent executable
        --output, -o name       : give a name to the compiled executable file
        --code                  : generate also intermediate .c and .h files
```

Both the logic description file (`lgi_file`) and the specification file (`spc_file`) may be noted without their suffixes. Default suffixes for ELAN files are: `.lgi` – for the top level logic description file, `.eln` – for modules used in the top level logic description file, `.spc` – for the specification file. The system ELAN searches these files (i.e. `.eln`, `.lgi`, `.spc`) in the following directories with the descending priorities:

- in the current directory,
- in the directory described by the value of a system variable $SECONDELANLIB$,
- in the directory described as $ELANLIB/commons$,
- in the directory described as $ELANLIB/noquote$, resp. $ELANLIB/quote$.

The value of the variable $ELANLIB$ can be locally replaced by the switch `--elanlib`. The default value of the variable $SECONDELANLIB$ is set to the parent directory, however, it can be also locally replaced by the switch `--secondlib` (or, `-l`) (for details, see below).

In Chapter 1, we have shown, how to simply run the ELAN interpretor with a logic description. Now, we illustrate on several examples the usage of different switches helping to the user more comfortable exploit all capabilities of the ' ELAN system.

The switch `-d`, (or, `--dump`) dumps out the signature of the described logic (i.e. all sorts and function symbols with their profiles), definitions of all strategies and rewrite rules. For the example `poly3` from Chapter 1,

```
> elan -d poly3 someVariables
```

we obtain a list of rewrite rules of the form:

```
local rule expand:poly/poly3[Vars] [1/1/fsym=214/vars=9]
 ( VAR(0)+(( VAR(1)+ VAR(2))*( VAR(3)+ VAR(4))))         =>
                    ( VAR(0)+( VAR(8)+( VAR(7)+( VAR(6)+ VAR(5)))))
     where VAR(5) := (simplify:poly/poly3[Vars]) ( VAR(2)* VAR(4))
     where VAR(6) := (simplify:poly/poly3[Vars]) ( VAR(2)* VAR(3))
     where VAR(7) := (simplify:poly/poly3[Vars]) ( VAR(1)* VAR(4))
     where VAR(8) := (simplify:poly/poly3[Vars]) ( VAR(1)* VAR(3))

local rule expand:poly/poly3[Vars] [2/2/fsym=214/vars=6]
 ( VAR(0)+(( VAR(1)+ VAR(2))* VAR(3)))   => ( VAR(0)+( VAR(5)+ VAR(4)))
     where VAR(4) := (simplify:poly/poly3[Vars]) ( VAR(2)* VAR(3))
     where VAR(5) := (simplify:poly/poly3[Vars]) ( VAR(1)* VAR(3))
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . e.t.c.
```

The names of rewrite rules (e.g. `expand`) are attached by a sort (e.g. `poly`), over which these rules work, and by a name of the module, where they were defined (e.g. `poly3[Vars]`). Additional informations about the internal code of the top-most function symbol of its left-hand side and the number of variables of each rule are displayed. Variables of rewrite rules are transformed into a unnamed form of `VAR(i)`, where i is an internal index of a variable.

The print-out continues with a list of strategies in the form:

```
strategy global last_simplify:poly/poly3[Vars]
 repeat* (
   dc(expand:poly )
 );
 repeat* (
   dc(factorise:poly )
 )

strategy global simplify:poly/poly3[Vars]
 repeat* (
   dc(expand:poly );
   dc(factorise:poly );
   dc(factorise:poly );
   repeat* (
     dc(factorise:poly )
   )
 )
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . e.t.c.
```

where the label of the strategy is composed from the name of the strategy, its sort and the module of the origin (similarly, as for named rules).

The description of the signature of the described logic is composed from a set of defined (and internal) sorts:

```
Sorts
intern int, ident, identifier, string, list[identifier],
intern ident, int, variable, intern string, bool,
poly,
```

and a list of function symbols of with their profiles in the following form:

```
Function symbols
 @                    : (variable)poly       pri 0 code 212;
 @                    : (int)poly            pri 0 code 213;
 @ '+' @              : (poly poly)poly      assocRight     pri 1 code 214;
'(' @ '+' @ ')'       : (poly poly)poly      pri 0 code 214;
 @ '*' @              : (poly poly)poly      assocRight     pri 2 code 215;
'(' @ '*' @ ')'       : (poly poly)poly      pri 0 code 215;
'deriv' '(' @ ',' @ ')': (poly variable)poly  pri 0 code 216;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . e.t.c.
```

The switch `-t` (or, `--trace`) allows to trace an ELAN program. The maximal deepness of tracing can be specified, like `-t 9`, however, by default (i.e. `-t`), it traces all. The number specified as the trace level counts matchings of a left-hand side, verifications of the conditional part of a rule, and evaluations of strategies applied over terms in local affectations. Thus, if the user wants to trace his/her rewrite derivation up to 5-th level, the trace argument should be specified as `-t 15`.

```
> elan -t poly3 someVariables
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . e.t.c.
enter query term finished by the key word 'end':
deriv(X*X,X) end

[] start with term :
    deriv((X*X),X)

       [reduce] start:
       [0]  deriv((X*X),X)
                  [reduce] start:
                  [0] (X*deriv(X,X))
                  [1] (X*1)
                  [2]  X
                  [reduce] stop :
              applying strategy 'simplify:poly/poly3[Vars]' on  X
                trying  'expand:poly' on        X
                fail of 'expand:poly'
                trying  'expand:poly' on        X
                fail of 'expand:poly'
            setting VAR(3) on  X
                  [reduce] start:
                  [0] (deriv(X,X)*X)
                  [1] (1*X)
                  [2]  X
                  [reduce] stop :
              applying strategy 'simplify:poly/poly3[Vars]' on  X
                trying  'expand:poly' on        X
                fail of 'expand:poly'
                trying  'expand:poly' on        X
                fail of 'expand:poly'
            setting VAR(4) on  X
                  [reduce] start:
                  [0] (X+X)
                  [reduce] stop :
              applying strategy 'simplify:poly/poly3[Vars]' on (X+X)
```

```
                trying  'expand:poly' on       (X+X)
                fail of 'expand:poly'
                trying  'expand:poly' on       (X+X)
                fail of 'expand:poly'
             setting VAR(5) on (X+X)
       [1] (X+X)
       [reduce] stop :
     trying  'expand:poly' on   (X+X)
     fail of 'expand:poly'
     trying  'expand:poly' on   (X+X)
     fail of 'expand:poly'
     trying  'factorise:poly' on        (X+X)
     fail of 'factorise:poly'
     trying  'factorise:poly' on        (X+X)
     fail of 'factorise:poly'
     trying  'factorise:poly' on        (X+X)
     fail of 'factorise:poly'

[] result term:
   (X+X)

[] end
```

The switch `-s` (`-S`, or `--statistics`) displays a brief (resp. a complete) version of statistics. These statistics contain the information about the running time of the last query, average speed in rule rewrites per second, the statistics of [named/unnamed] rules [applied/tried].

```
> elan -S poly3 someVariables
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . e.t.c.
 enter query term finished by the key word 'end':
 deriv(X*X,X) end

[] start with term :
    deriv((X*X),X)

[] result term:
   (X+X)

[] end

 Statistics:
 total time     (0.003+0.000)=0.003 sec (main+subprocesses)

 average speed  1666 inf/sec
                5 nonamed rules applied,    23 tried
                0   named rules applied,    11 tried
 named rules
           applied   tried    rule for symbol
                0       8    expand:poly
                0       3    factorise:poly
 nonamed rules
           applied   tried    rule for symbol
                0       4    ( poly + poly )
                2      12    ( poly * poly )
                3       7    deriv( poly , variable )
 end of statistics
```

The complete statistics shows a detailed list of applications and tries for named and nonamed rules. The brief statistics do not precise the last paragraph about unnamed rules.

The switch `-w` (or, `--warningsoff`) eliminates all ambiguity warnings produced during parsing of the ELAN description of a logic. The switch `-q` (or, `--quiet`) suppresses all messages during the execution of the ELAN program. The switch `-b` (or, `--batch`) suppresses all messages, thus this mode is useful when the ELAN system is executed from a script file (in the batch mode).

The switch `--elanlib dir` locally redefines the value of the system variable *ELANLIB* by the string `dir`. The switch `-l dir` (`--secondlib dir`) does the same with the variable *SECONDELANLIB*.

The switch `-C` runs the system ELAN with a simple command language interpreted on the top-most level. This mode is described in the following section.

The switches `-c` (`--compiler`), `-O` (`--optimise`), `--code`, `-o` (`--output`), `-n` (`--deterministic`) and `--exe` concern with the ELAN compiler and they will be explained in Chapter 2.4

After loading all modules specified in the logic description file, ELAN requires to enter a query term, which has to be finished by the word 'end' or by character `^D` (control-D). Its evaluation can be interrupted by typing a `^C`, the interpreter then proposes a simple run-time menu:

- **ExecutionAbort** - A - aborts the current execution and allows to enter another query term,
- **Continue** - C - continues interrupted execution,
- **Dump** - D - dumps the signature, rules and strategies of the currently loaded logic,
- **Exit** - E - quits the ELAN system,
- **Statistics** - S, s - writes the actual statistics corresponding to the last evaluated query,
- **ChangeTrace** - T - changes trace level of the current derivation such that the interpreter asks for a new trace level,
- **ChangeQuiet** - Q - switches between two displaying modes: quiet – unquiet,

2.3 Top level interpretor

The command language of ELAN slightly improves the user's interface such that it offers to the user several commands to:

- change several parameters of the system,
- debug your programs using break-points,
- keeping the history of your queries and results,
- runs loaded logic with different strategies, different entries, etc.

Starting the system ELAN with the option `-C`, ELAN instead of asking us to
`enter query term finished by the key word 'end'`
asks us to
`enter command finished by ';'`
We show a brief description of all commands (also found in the file *ELANLIB/help.txt*), which are later illustrated on several examples.

quit terminates the session and leave the system ELAN.

help shows the help file *ELANLIB/help.txt*,

load mod loads (imports) an additional module, where `mod` (an identifier) describes its name with their arguments,

batch mod calls a script file with the name `mod` (an identifier) containing a stream of commands of the command language. It tries to evaluate all commands of this script file, and then, it returns the control to the interactive mode. The script files can be concatenated.

qs shows the queue of input queries (history of the last 10). Sorted terms in this queue of queries **qs** can be referenced by identifiers `Q`, `Q1`, ..., `Q9` respecting their sorts. These sorted identifiers could be used for the construction of further queries.

rs shows the queue of the latest results (also, history of the last 10). Terms in the queue of results **rs** are referenced by identifiers `R`, `R1`, ..., `R9`. In the case of the non-deterministic computation, there is not a correspondence between the query term `Qi` and the result term `Ri`.

startwith (str)term redefines the 'start with' term originally defined in the logic description file `.lgi`. The term `term` may contain also symbols `query` standing for a place-holder of an input term entered by the user (by the command `run`), and/or `Q`, ..., `Qi` or `R`, ... `Ri` standing for values of the previous queries or results.

checkwith term redefines the 'check with' term originally defined in the logic description file. The boolean term `term` may contain the any symbols `query`, `Q`, ..., `Qi` or `R`, ... `Ri`.

printwith term sets-up the 'print-with' term. The 'print-with' term may contain the symbol `result`, which is a place-holder for result terms. Intuitively, the meaning of the 'print-with' term P is, that each result r of the computation is substituted for the place-holder `result` in the term P. The substituted term $P[result/r]$ is normalised and printed out. This command can be used to an automatic transformation of results into a different form, because of different presentation formalism. An application of this could be a transformation of very large result terms into more readable (and, maybe, not complete) notation, or a conversion into a LaTeXnotation. The default value of the 'print-with' term is `result`.

run term evaluates the term `term` as a query w.r.t. the logic imported to the system ELAN, eventually returns results. The entry term `term` may contain identifiers `Q`, `Q1`, ..., `Q9`, (resp. `R`, `R1`, ..., `R9`) referring items of the queue of queries **qs** (resp. results **rs**).

sorts type type type sets-up sorts of the symbols `query`, `result` and of the 'print-with' term.

stat prints the statistics,

dump prints rules, strategies and the signature,

dump name dumps only named rules with the name `name`, strategies named by `name` or unnamed rules with the head symbol `name`.

dump n dumps only unnamed rules of a function symbol with the internal code `n`.

display n if `n` is not zero, all printed terms are in the internal form. The internal form of a term attaches to a name of a symbol also its internal code, which allows to identified overloaded function symbols. **display 0** switches print-outs into traditional (thus, overloaded) form.

trace n sets-up the trace level to the value `n`.

break name (resp. **unbreak name**) sets (resp. removes) break-points to all named rules with the name `name`, to all strategies named by `name` and to all unnamed rewrite rules with the head symbol `name`.

break n (resp. **unbreak n**) sets (resp. removes) break-points to unnamed rules with the head symbol with the internal code `n`.

breaks shows a list of all breakpoints.

We illustrate the command language on a small ELAN session with the example `poly3` from the previous Chapter. When the switch `-C` is specified, the system ELAN loads the module `Query` describing the syntax of commands.

```
> elan -q -C poly3 someVariables

        ******** ELAN version 2.00 (17/04/97.14:08) ********

          (c)   INRIA-Lorraine  &  CRIN,   1994, 1995, 1996, 1997
   handling module identifier
     handling module ident
       handling module bool
       end of bool
     end of ident
     handling module eq[identifier]
       handling module cmp[identifier]
       end of cmp[identifier]
     end of eq[identifier]
   end of identifier
   handling module list[identifier]
     handling module int
     end of int
   end of list[identifier]
   handling module poly3[Vars]
     handling module eq[variable]
       handling module cmp[variable]
       end of cmp[variable]
     end of eq[variable]
   end of poly3[Vars]
 handling module Query[poly,poly,poly]
 end of Query[poly,poly,poly]
```

We can run simple queries using the command `run`:

```
enter command finished by ';':
run deriv(X*X,X);
[] start with term :
    deriv((X*X),X)
[] result term:
   (X+X)
[] end

enter command finished by ';':
run deriv(X*X*X+X*X+X+1,X);
[] start with term :
    deriv(((X*(X*X))+((X*X)+(X+1))),X)
[] result term:
   (1+(X*(((X*2)+2)+X)))
[] end
```

The queries and their results are stored in two queues, which can be listed by commands `qs`, `rs`:

```
enter command finished by ';':
qs;
Queries ... 2 : elements
Q1      poly     deriv((X*X),X)
Q       poly     deriv(((X*(X*X))+((X*X)+(X+1))),X)

enter command finished by ';':
rs;
Results ... 2 : elements
R1      poly    (X+X)
R       poly    (1+(X*(((X*2)+2)+X)))
```

The previous queries and their results can be referenced for the construction of the new ones:

```
enter command finished by ';':
run deriv(R,X) ;
[] start with term :
    deriv((1+(X*(((X*2)+2)+X))),X)
[] result term:
   ((X*6)+2)
[] end

enter command finished by ';':
run deriv(Q*Q,X) ;
[] start with term :
    deriv((deriv((1+(X*(((X*2)+2)+X))),X)*deriv((1+(X*(((X*2)+2)+X))),X)),X)
[] result term:
   (24+(X*72))
[] end
```

Function symbols defined by unnamed rules can be debugged, for example, if we know their internal code. In the following example, the command `dump` displays the symbol with the internal code `216`, and consequently, we put a break point on this function.

```
dump 216;
function dump
'deriv' '('  @ ','  @ ')'        : (poly variable)poly   pri 0 code 216;

enter command finished by ';':
break 216;
function dump
'deriv' '('  @ ','  @ ')'        : (poly variable)poly   pri 0 code 216;
```

Having break-points set, the execution shows all entry and exit points of traced functions, or strategies:

```
enter command finished by ';':
run deriv(X*X,X);

[] start with term :
    deriv((X*X),X)
 [0]  deriv((X*X),X)
       [0]  deriv(X,X)
       [1]  1
       [0]  deriv(X,X)
       [1]  1
 [1] (X+X)
[] result term:
   (X+X)
[] end
```

We can switch all outputs into the internal form by the command `display 1`:

```
enter command finished by ';':
display 1;

enter command finished by ';':
run deriv(X*X,X);
[] start with term :
    deriv_216((X_209*X_209),X_209)
 [0]  deriv_216((X_209*X_209),X_209)
       [0]  deriv_216(X_209,X_209)
       [1]  1
       [0]  deriv_216(X_209,X_209)
       [1]  1
 [1] (X_209+X_209)
[] result term:
   (X_209+X_209)
[] end
```

We can change the 'start-with' term to `(last_simplify)deriv(query,X)`, and the command `stat` informs us about the current setting and the performance of the last executed query.

```
enter command finished by ';':
startwith (last_simplify)deriv(query,X);

enter command finished by ';':
stat;
Input type : poly
Output type: poly
Print type : poly
Strategy   : last_simplify
Start_with :  deriv_216( VAR(0),X_209)
Check_with :  true_1
Print_with :  VAR(0)
------------

 Statistics:
 total time     (0.009+0.000)=0.009 sec (main+subprocesses)

 average speed  555 inf/sec
                5 nonamed rules applied,    23 tried
                0   named rules applied,    11 tried
 named rules
           applied   tried    rule for symbol
                 0       8    expand:poly
                 0       3    factorise:poly
 nonamed rules
           applied   tried    rule for symbol
                 0       4    ( poly + poly )
                 2      12    ( poly * poly )
                 3       7    deriv( poly , variable )
 end of statistics
```

2.4 How to run the ELAN compiler

The ELAN compiler transforms a logic description into an executable binary file. The compiler (for detailed description, see [Vit96]) produces an efficient `C++` code, which is later compiled by `gnu g++` compiler. The executable binary code is dependent on the particular architecture, because a small part of the ELAN library performing the non-determinism, which is written in

assembler language. Up to now, this is the only known limitation, why the compiler runs only on the following architectures: DEC-ALPHA, SUN4 and Intel-PC. The ELAN compiler does not (in the alpha release of the version 2.0) compile whole ELAN language. There are several fine restrictions (see Section 4.3) of programs, which can be compiled. The main restriction is that the ELAN compiler does not treat AC-symbols, thus any logic description containing AC-symbols is not yet compilable.

In the preliminary alpha release of the version 2.00, there exists two modes of using the ELAN compiler. The first one (initialised by the switch `-c`), allows to compile a logic description (or, an ELAN program) with an input query term into a binary executable file. This file is launched from the ELAN environment, and also their results are interpreted by the ELAN environment such that the user does not see any substantial difference in the behaviour of the ELAN environment between the interpreted and the compiled versions (except of the speed-up).

We illustrate the compiler on a simple example from the distribution file. The first example shows the compilation of the program `nqueens` with the query `queens(8)`:

```
> elan -c nqueens

        ******** ELAN version 2.00 (17/04/97.16:24) ********

          (c)   INRIA-Lorraine  &  CRIN,   1994, 1995, 1996, 1997
   handling module nqueens
     handling module int
       handling module bool
       end of bool
     end of int
   end of nqueens
[.eln->.c]:
[.c->.o]: $ELANLIB/Compiled/`uname -m`/cElanScript -c  nqueens

gcc -DNONUNDERSCORED -w -I/users/protheo/toto/elan/elanlib//Compiled
    -L/users/protheo/toto/elan/elanlib//Compiled/alpha -c nqueens.c

enter query term finished by the key word 'end':
queens(8) end
[.o->a.out]: $ELANLIB/Compiled/`uname -m`/cElanScript -l  nqueens -N

gcc -DNONUNDERSCORED -w -I/users/protheo/toto/elan/elanlib//Compiled
    -L/users/protheo/toto/elan/elanlib//Compiled/alpha maincompiled.c
    nqueens.o -lelan -lnonearley -lm
[execute]:
  [ main ] start :
  [] result term :5.7.2.6.3.1.4.8.nil
  [] result term :4.7.5.2.6.1.3.8.nil
  [] result term :6.4.7.1.3.5.2.8.nil
  [] result term :6.3.5.7.1.4.2.8.nil
  [] result term :4.2.8.6.1.3.5.7.nil
  [] result term :5.3.1.6.8.2.4.7.nil
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . e.t.c.
  [] result term :3.5.2.8.6.4.7.1.nil
  [] result term :5.2.4.7.3.8.6.1.nil
  [] result term :4.2.7.3.6.8.5.1.nil
   real 0.3 user 0.1 sys 0.0
[kill]:
 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . e.t.c.
enter query term finished by the key word 'end':
```

The same program (but without the query) can be compiled using the option `--exe`, which produces an independent executable file. This fact implies that the parser of ELAN terms is

included into the stand alone executable file.

```
> elan --exe nqueens

        ******** ELAN version 2.00 (17/04/97.22:31) ********

          (c)   INRIA-Lorraine  &  CRIN,   1994, 1995, 1996, 1997
   handling module nqueens
     handling module int
       handling module bool
       end of bool
     end of int
   end of nqueens
[.eln->.c]:
[.c->a.out]: gcc -O2 -DNONUNDERSCORED -w -I/users/protheo/toto/elan/elanlib//Compiled
    -L/users/protheo/toto/elan/elanlib//Compiled/`uname -m`
    -DEARLEY nqueens.c -o a.out -lelan -learley -lm
/bin/rm -f nqueens.c nqueens.h
```

This executable file `a.out` is independent of the input query, and thus it asks the user the input term over which the execution is launched.

```
> a.out
Query:
queens(8) end

[] result term:
     5.7.2.6.3.1.4.8.nil
[] result term:
     4.7.5.2.6.1.3.8.nil
[] result term:
     6.4.7.1.3.5.2.8.nil
[] result term:
     6.3.5.7.1.4.2.8.nil
[] result term:
     4.2.8.6.1.3.5.7.nil
```

There are several supplementary switches for the stand alone compiling with the switch `--exe`:

The switch `--code` does not remove the temporary files `nqueens.c` and `nqueens.h` produced by the ELAN compiler and passed to `g++`.

The switch `-o nqueens` (or, `--output nqueens`) produces the executable file with the name `nqueens`.

The switch `-O` passes the optimisation flag the `g++` compiler.

The switch `--deterministic` (or, `-n`) use the deterministic run-time library, which is independent on the architecture. This allows to run the ELAN compiler only with deterministic programs (very roughly saying, not using ELAN strategies).

More detailed description of the compiler can be found in Chapter 4.3.

Chapter 3

ELAN: the language

This chapter presents a full bottom-up description of the language features in the sense that in opposition to what we have done in Chapter 1, a construction of the language is used only if it has been introduced before.

All examples given below are running under the current implementation of the ELAN version V2.00.

3.1 Lexicographical conventions

3.1.1 Separators

All ASCII characters whose code are less or equal than 32 are considered by the parser as *separators*. Thus spaces, tabulations and newlines are separators.

3.1.2 Lexical unities

A *lexical unity* in ELAN is either an identifier, a natural number or a special character.

Identifiers are composed by concatenation of letters, numerals and the character "_", and they should begin by a letter.

<*letter*> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
| a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<*numeral*> ::= 0|1|2|3|4|5|6|7|8|9

<*identifier*> ::= <*letter*> { <*letter*> | <*numeral*> | _ } *

A *number* is the concatenation of numerals.

<*number*> ::= { <*numeral*> } +

All characters different from the previously introduced ones (letters, numerals, separators) and the characters '{', '}', '~', are considered as special characters and are referenced as <*char*> .

In function symbol names, we are also using the lexical unity <*nchar*> which corresponds to any character in <*char*> except: '@' and ':'.

3.1.3 Comments

All strings enclosed between the strings "`/*`" and "`*/`" or between the string "`//`" and the end of line, are considered by the parser as *comments*.

3.2 Element modules

Element modules are the basic unit blocks of the language. They allow to define computational systems i.e. to define a rewrite theory together with a strategy. For modularity reasons, importations are made possible in ELAN modules, under the condition that no cycle exists in the importation relation. This is described in the modularity section 3.7.

The syntax is the following:

<module> ::= **module** *<formal module name>*
[*<imports>*]
[*<sort definition>*]
[*<operator definition>*]
{ *<family of rules>* } *
{ *<family of strategies>* } *
end

The module of name `ModuleName` should be declared in the file `ModuleName.eln`. Only one, and exactly one module can be described in a given file.

Importations can be made local or global:

<imports> ::= **import**
[**global** { *<module name>* } + **;**]
[**local**] { *<module name>* } + **;**
end

Sorts are always global:

<sort definition> ::= **sort** { *<sort name>* } + **; end**

but operators can be exported, and thus declared global, or just local in which case they can be used only in the module where they are defined.

<operator definition> ::= **operators**
[**global** { *<symbol declaration>* **;** } +]
[**local** { *<symbol declaration>* **;** } +]
end

<module name> ::= *<sort name>*

3.3 Definition of signatures

ELAN is a multi-sorted language, and signatures are composed by the definitions of the sorts and operators symbols. The operator syntax is given in a mix-fix form which allows the super-user to define very conveniently its own syntax.

3.3.1 Sort declarations

A *sort declaration* is either a sort name (an identifier) or a sort name followed by the list of sort names it depends on:

<sort name> ::= *<identifier>*
| *<identifier>* [*<sort name>* { , *<sort name>* } *]

Example 3.1 bool, int, list[bool], pair[int,list[bool]] *are sort names in* ELAN.

Remark: A sort name is always *global* in ELAN. Two sort declarations using the same sort name, will result in the creation of only *one* sort.

Built-in sorts For efficiency purpose the following sorts are built-in in ELAN:

1. bool which refers to the standard booleans. The ELAN module `bool` that introduces this sort is defined on page 59.
2. ident which refers to any identifier. The ELAN module `ident` that introduces this sort is defined on page 60.
3. int which refers to the standard integers. The ELAN module `int` that introduces this sort is defined on page 60.
4. double which refers to the "standard" floating point numbers provided by the C language. The ELAN module `double` that introduces this sort is defined on page 60.

3.3.2 Function declarations

A *function symbol* (also called *operator* below) is given either by defining a new symbol together with rank or by aliasing an existing symbol, optional properties can be specified:

<symbol declaration> ::= *<new symbol declaration>*
| *<alias>*

<new symbol declaration> ::= *<identifier>* : *<rank>* [*<options>*]

<alias> ::= *<new symbol declaration>* **alias** *<old name>* :

In case of *aliasing*, *<new symbol declaration>* is only another name for the function symbol *<old name>* . The *two* names are then accessible and refer to the *same* object.
▷ The last alias defined is considered by the parser as having the highest priority with respect to the previously defined ones.

The name of a function symbol contains information that will be used by the mixfix parser to read a term:

<name> ::= { *<symbol>* } +

<symbol> ::= *<single lexem>*
| @
| ' *<single lexem>* '
| '\ *<number>* '

<single lexem> ::= *<identifier>*
| *<number>*
| *<nchar>*

where *<identifier>* is any identifier except keywords. Notice that indeed keywords can be used without their special meaning when placed between quotes.

Similarly, *<nchar>* can be any character except letters, digits and:

{ } ~ : @

The characters ' {, }, ~' are restricted to the pre-processor use. Thus they cannot be used in any module, but they can be used in the specification file or in the description of the query, since both are not pre-processed. The ':' character indicates the end of the name and the character '@' is the place holder (the underline character in OBJ). These two last characters can be used with another semantics than their built-in one, when placed between quotes.

A quoted number after the character '\' represents the definition of a character by its ascii code. This facility can be used in order to add to the syntactic form of a term non-visible characters or the characters ' {, }, ~'.

Finally an operator declaration is achieved by defining its *rank*, which is just a list of sort names:

<rank> ::= *<sort name>*
| ({ *<sort name>* } +) *<sort name>*
| ({ *<name>* : *<sort name>* } +) *<sort name>*

Example 3.2 *With the two declarations:*

```
@ + @                : ( int int ) int
+(@,@)               : ( int int ) int                    alias @+@:
```

the strings "`x + y`*" et "*`+(x,y)`*" represent the same term.*

When defining a constructor operator, it is possible to define the related selectors and modifiers, as outlined in the next example.

Example 3.3 *In the following module example, the selector* `first` *and* `second` *are specified and the system will automatically generate the corresponding selector and modifier operators and the corresponding rewrite rules.*

```
module Pair                                                          1
import bool;                                                         2
 end                                                                 3
sort X Y Pair;                                                       4
 end                                                                 5
operators                                                            6
 global                                                              7
 [@,@]              :  ( first:X second:Y ) Pair;                    8
 isPair(@)          :  ( Pair ) bool;                                9
 end                                                                 10
end                                                                  11
```

This is similar to the following module:

```
module pair                                                          1
import bool;                                                         2
 end                                                                 3
sort X Y Pair;                                                       4
 end                                                                 5
operators                                                            6
```

```
global                                                                  7
[@,@]              :  ( X Y ) Pair;                                     8
isPair(@)          :  ( Pair ) bool;                                    9
          automatically generated selectors                             10
@ .first :  (Pair) X;                                                   11
@ .second :  (Pair) Y;                                                  12
          automatically generated modifiers                             13
@[.first<-@] :  (Pair X) Pair;                                          14
@[.second<-@] :  (Pair Y) Pair;                                         15
                                                                        16
end                                                                     17
end                                                                     18
                                                                        19
```

where the following rewrite rules are in fact automatically generated:

```
(x,y).first => x
(x,y).second => y
(x,y)[.first<-z] => (z,y)
(x,y)[.second<-w] => (x,w)
```

3.3.3 Function declaration options

As in languages like OBJ or ASF+SDF, an operator declaration carries a lot of information about the syntax and the semantics of the operator. We also need to specify complementary information like the precedence of the operator for the parser, or some of its semantic properties like associativity. This is done in ELAN using the following syntax:

<option> ::= **assocLeft**
| **assocRight**
| **pri** *<number>*
| **(** **AC** **)**
| **code** *<number>*

Parsing Associative operators The first two options are useful for defining syntactic associativity of symbols:

Example 3.4 *The declarations:*

```
@ * @              : ( int int ) int      assocLeft
*(@,@)             : ( int int ) int                        alias @*@:
```

allow to parse the term $x * y * z$ *as* $*(*(x,y),z)$.

Priorities The *priority* parsing problems can be solved using the priority option:

Example 3.5 *The declarations:*

```
@ + @              : ( int int ) int      assocLeft pri 10
@ * @              : ( int int ) int      assocLeft pri 20
```

allow to parse an expression like $x + y * z$ *as usual in* $x + (y * z)$.

Coercions can be defined and even hidden as in the following example:

Example 3.6 *Assume that equations are constructed with the equality symbol:*

```
@ = @              : ( term term ) equation;
```

and equational systems as conjunctions of equational systems:

```
@ & @              : ( eqSystem eqSystem ) eqSystem;
```

An equation should be considered as an equational system, which corresponds to consider the sort `equation` *as a subsort of* `eqSystem`*. This is done by the declaration of an invisible operator:*

```
@                    : ( equation ) eqSystem;
```

This (invisible) coercion allows parsing the expression `(P & e)` *where* `P:eqSystem; e:equation`.

Semantic properties Semantic properties (currently only associativity and commutativity (abbreviated *AC*)) could be used as follows. Note that in this case AC matching is used during evaluation. This can lead to efficiency problems, since AC-matching is NP-complete in case of non-linear patterns [BKN87].

Example 3.7 *One can define a union operator which is associative and commutative in the following way:*

```
@ ∪ @                : ( set set ) set      (AC)
```

User built-in operators Finally, in order to improve efficiency of the evaluation, built-in operators can be specified using the **code** option. In this case the natural number specified after the **code** keyword specifies the index of this symbol in the symbol table of ELAN.

Example 3.8 *The symbol* `true` *is defined in the* `bool` *module as the first symbol of the symbol table:*

```
true                 : bool                 code 1
```

See also the `bool` *definition module on page 59.*

Section 5.1.5 summarized the codes already defined for ELAN's standard built-ins.

▷ Adding built-in operators is of course a delicate matter and we do not recommend to do so without having access to the ELAN source code.

3.3.4 Built-in function declarations

For efficiency reasons, the following operators are built-in in ELAN:

1. `true` and `false`. This is deeply used in the implementation since the evaluation of conditional rules uses the `true` value.

2. On integers, the standard arithmetic functions are built-in. We recommand to use these functions only through the interface of the ELAN module `int`.

3. On built-in identifiers, the equality functions are defined. We recommand to use these functions only through the interface of the ELAN module `identifier`.

4. `eq`, `occur` et `replace`. They realize respectively polymorphic operators for equality, testing subterm and the replacement of a subterm by another one. Their full descriptions are given on page 59, 60 and 60.

5. All built-in operators are described in the Table 5.1.

3.4 Definition of rules

Rewriting is the basic concept underlying the design of ELAN. The language allows defining labeled conditional rules with local assignments. As it will be emphasized in section 3.6 on the evaluation mechanism, there is a *very important* difference between:

– **labelled rules** which evaluation is fully controled by the user defined strategies and

– **non-labelled rules** which are performing functional evaluation.

A *condition* controling the firing of a rule is a term of sort `bool` introduced by the **if** keyword.
A *local assignment* is introduced by the keyword **where** and it assigns to a variable the result of the application of a strategy to a term.

3.4.1 Rule syntax

Rules are introduced in families, possibly restricted to a single element.

<family of rules> ::= **rules for** *<sort name>*
[{ *<variable declare>* } +]
[**global** { *<labelled or non-labelled rule>* } +]
[**local** { *<labelled rule>* } +]
end

<labelled or nolabelled rule> ::= *<labelled rule>*

| *<non-labelled rule>*

<labelled rule> ::= [*<rule label>*] *<rule body>* **end**

<rule label> ::= *<identifier>*

<non-labelled rule> ::= [ ] *<rule body>* **end**

<rule body> ::= *<term>* => *<rhs>*

<rhs> ::= *<term>*
| *<rhs>* **if** *<boolean term>*
| *<rhs>* **where** *<var name>* := ([*<strategy>*]) *<term>*
| *<rhs>* **where** *<var name>* := [*<Pstrategy>*] *<term>*
| *<rhs>* **where** (*<sort>*) *<term>* := ([*<strategy>*]) *<term>*
| *<rhs>* **where** (*<sort>*) *<term>* := [*<Pstrategy>*] *<term>*
| **switch**
{ **case** *<boolean term>* **then** *<rhs>* } +
[**otherwise** *<rhs>*]
end

Example 3.9 *Here is one way for defining the idempotency rules for booleans. Notice that the two rules are local to the module so that their label can only be used for defining strategies in this module. Notice also that the two rules have, on purpose, the same label:*

```
rule for boolean                                                    1
          x :  boolean;                                             2
local                                                               3
[idempotency] x and x => x                                          4
end                                                                 5
                                                                    6
rule for boolean                                                    7
          x :  boolean;                                             8
local                                                               9
[idempotency] x or x => x                                           10
end                                                                 11
```

Since several rules may have the same label, the resulting ambiguities are handled by the strategies. The rule label is optional and rules may have no name. This last case is designed in such a way that the intended semantics is functional, and it is the responsability of the rewrite rule designer to check that the rewrite system is confluent and terminating. Unlabeled rules are handled by a built-in strategy (leftmost-innermost) as it is fully described in Section 3.6.

3.4.2 The where construction

The **where** construction has several purposes. The first one is to modularize the computations by preparing them and affecting the result to variables local to the rule. The second is to direct the evaluation mechanism using strategies and the built in left-most inner-most evaluation reduction. Let us review the possibilities offered by ELAN.

The **where** statement is first useful when one wants to call a strategy on subterms.

Example 3.10 *As a first introducing example, let us consider the module:*

```
module ruleWithWhere                                                1
                                                                    2
sort foo; end                                                       3
                                                                    4
operators global                                                    5
          a :  foo;                                                 6
          b :  foo;                                                 7
 end                                                                8
                                                                    9
rules for foo                                                       10
          x :  foo ;                                                11
global                                                              12
 [r0]     a => b                                 end                13
 [r1]     a => x where x := ()b                  end                14
 [r2]     a => x where x := (strat)b             end                15
 end                                                                16
end                                                                 17
```

Using the rule `r0`*, the term* `a` *reduces in one step to* `b`*. Using the rule* `r1`*, the term* `b` *is first normalized using non-labelled rules and then* `a` *is replaced by* `b`*. Using the rule* `r2`*, the term* `b` *is first normalized into* `b'` *using non-labelled rules and then the strategy* `strat`*, defined elsewhere by the super-user, is applied on* `b'` *and the result is substituted to* `a`*.*

One can also perform more complex affectations of the local variables by using patterns.
— Release Note: The syntax used to define the sort of a pattern is not still appropriate (confusion with strategy call is possible), so this will change is the beta version.

The current syntax of the **where** statement with patterns can be rephrased as follows:

$$\textbf{where } (sort)pat \ ::= ()term \mid (s)term \mid [s]term$$

where the pattern *pat* is of the sort *sort* and is assumed to be a term of sort s. AC-operators as well as non-linearity are allowed for patterns.

This pattern in where capability is examplified in the next subsection.

3.4.3 Switches

When specifying with conditional rewrite rules, one of the syntactic difficulty is to write a family of rewrite rules in such a way that the disjunction of the conditions covers all the possible cases. The **switch** construction in ELAN intend to facilitates such a writing.

The syntax of the **switch** construction can be rephrased as follows:

$$\begin{array}{ll} l \Rightarrow & \textbf{switch} \\ & \quad whifs \\ & \quad \textbf{case } c_1 \textbf{ then } r_1 \quad whifs_1 \\ & \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ & \quad \textbf{case } c_n \textbf{ then } r_n \quad whifs_n \\ & \quad \textbf{otherwise} \quad r_0 \quad whifs_0 \\ & \textbf{end} \\ \textbf{end} & \end{array}$$

where the sequence of wheres and ifs is defined as follows:

$$whifs ::= \{ \textbf{ where } affectation \mid \textbf{ if } c\}^*$$

As expected, the semantics of the **switch** construction is the following:

$$\begin{array}{llll} l \Rightarrow & r_1 \quad whifs & \textbf{if } c_1 & whifs_1 \\ l \Rightarrow & r_2 \quad whifs & \textbf{if } c_2 \textbf{ if } not(c_1) & whifs_2 \\ & \ldots \quad \ldots & \ldots \quad \ldots & \\ l \Rightarrow & r_n \quad whifs & \textbf{if } c_n \textbf{ if } not(c_{n-1}) \textbf{ if } not(c_1) & whifs_n \\ l \Rightarrow & r_0 \quad whifs & & whifs_0 \end{array}$$

which means that:
— either c_1 is true in which case l rewrites to r_1,
— or c_1 is false and c_2 is true in which case l rewrites to r_2,
— or c_1 and c_2 are false and c_3 is true in which case l rewrites to r_3,
— ...
— else l rewrites to r_0.
The construction **switch** is recursive.

Example 3.11 *This example shows the use of patterns as well as switch.*

```
module quick                                                    1
import global int list[int]; end                                2
sort pair; end                                                  3
operators global                                                4
```

```
 sort(@) :  (list[int]) list[int];                                          5
 pivot(@,@,@,@) :  (int list[int] list[int] list[int]) pair;                6
 [@,@] :  (list[int] list[int]) pair;                                       7
end                                                                         8
                                                                            9
rules for list[int]                                                         10
x :  int;                                                                   11
xs,s,l :  list[int];                                                        12
global                                                                      13
 [] sort(nil) => nil                                                        14
 end                                                                        15
 [] sort(x.xs) => append(sort(s),x.sort(l))                                 16
 where (pair)[s,l]:=()pivot(x,xs,nil,nil)                                   17
 end                                                                        18
end                                                                         19
rules for pair                                                              20
p, x :  int;                                                                21
xs,s,l :  list[int];                                                        22
global                                                                      23
 [] pivot(p,nil,s,l) => [s,l]                                               24
 end                                                                        25
 [] pivot(p,x.xs,s,l) => switch                                             26
 case p < x then pivot(p,xs,s,x.l)                                          27
 otherwise pivot(p,xs,x.s,l)                                                28
 end                                                                        29
 end                                                                        30
end                                                                         31
end                                                                         32
```

3.4.4 Labels visibility

The **global** and **local** attributes make sense only for labelled rules. When a labelled rule is:

– **local** then its label can only be used in the module in which the rule is declared,

– **global** then its label can be used in all the module importing the module defining the rule, with the visibility described in the section on modularity (see section 3.7).

▷ Because of their semantics, non-labelled rules are always global.

Remark: We will see next on page 51 how rule families can be constructed automatically in ELAN using the pre-processor construction "for each".
— Release Note: In this version release, the notion of single rule has been removed and the notion of rule locality has been introduced, so that rules labels can now be declared to be local to a module or on the contrary globally accessible. See section 3.10 for details concerning the changes.

3.5 Definition of basic strategies

The notion of strategy is one of the main originality of ELAN. Practically, a strategy is a way to describe which rewrite computations the user is explicitly interested in. It specifies where and when a given rule should be applied in the term to be reduced. From a theoretical point of view, a strategy is a subset of all proof terms defined by the current rewrite theory. The application of a strategy to a term results in the (possibly empty) collection of all terms that can be derived from the starting term using this strategy [KKV95, Vit94]. When a strategy returns an empty set of terms, we say that it *fails*.
▷ We distinguish (at least historically) between *basic strategies* and *PStrategies*. The first ones are described in this section and are mainly based on regular expressions, while the latter allow the user

to define quite elaborated strategies and are described latter

— Release Note: PStrategies are not described in this version.

ELAN allows to define basic strategies in two steps. The *first level* consists of defining regular expressions built on the alphabet of rule labels. But a rule can be applied using a user-defined strategy only at the *root* of a term. Thus this is combined with a *second level* that consists of using strategies in the **where** construction of rule definitions. We will see through examples that the expressive power of strategies in ELAN is far more than just regular expressions and that, because of the second level, rules can indeed be applied everywhere in a term.

3.5.1 Basic strategies syntax

The general syntax of basic strategies is the following:

```
<family of strategies> ::= strategies for <sort name>
                           [ global { <strategy object> } + ]
                           [ local { <strategy object> } + ]
                           end

<strategy object> ::= [ <strategy label> ] <strategy> end

<strategy> ::= <strategy label>
             | <choosing>
             | <concatenation>
             | <iterator>
             | <normalize>
             | fail
             | id

<strategy label> ::= <identifier>

<choosing> ::= dont care choose ( <strategy_rule> { || <strategy_rule> } * )
             | dont know choose ( <strategy_rule> { || <strategy_rule> } * )
             | dc ( <strategy_rule> { || <strategy_rule> } * )
             | dk ( <strategy_rule> { || <strategy_rule> } * )
             | first ( <strategy_rule> { || <strategy_rule> } * )

<strategy_rule> ::= <strategy>
                  | <rule label>

<concatenation> ::= <strategy> ; <strategy>

<iterator> ::= iterate ( <strategy> )
             | repeat* ( <strategy> )
             | repeat+ ( <strategy> )

<normalize> ::= normalize ( <strategy> )
              | normalise ( <strategy> )
```

3.5.2 Handling one rule

The application of a rewrite rule in ELAN yields, in general, several results: i.e. there are in general several ways to apply a given conditional rule with local assignments. This is first due to equational matching (e.g. AC-matching) and second to the **where** assignment, since it may itself recursively return several possible assignments for variables, due to the use of strategies.

Thus the language provides basic constructions to handle this non-determinism. This is done using the basic strategy operators: **dont care choose** and **dont know choose**.

For a rewrite rule $\ell : l \to r$ the strategy **dont care choose**(ℓ) returns *at most one* result which is undeterministically taken among the possible results of the application of the rule.

▷ The current implementation returns the first one, but this is not part of the operational nor mathematical semantics and this may change is future version of the system. The first strategy on the contrary returns always the first one.

On the contrary, if the ℓ rule is applied using the **dont know choose**(ℓ) strategy, then *all* possible results are computed and returned by the strategy. The implementation handles these several results by an appropriate back-chaining operation.

▷ A strategy enumerates all terms it describes, should this collection be finite or not. Consequently the user should note that in case (s)he writes a strategy that enumerates an infinity of terms, then the evaluation process will of course not terminate.

Example 3.12 *Let us consider the following incomplete module:*

```
empty      :                  set;                                          1
@          :  ( elem )        set;                                          2
@ U @      :  ( set set )     set       (AC);                               3
element(@):  ( set )          elem;                                         4
                                                                            5
                                                                            6
rules for elem                                                              7
          S : set;                                                          8
          e :  elem;                                                        9
local                                                                       10
 [extractrule] element(S U e) => e                  end                     11
end                                                                         12
```

The application of the rule `extractrule` *on the term* `empty U a U b` *can be done in two ways, since the AC-matching algorithm returns a complete set of matches consisting in the two substitutions:* { `S` ↦ `empty U a, e` ↦ `b`} *and* {`S` ↦ `empty U b, e` ↦ `a`}.

When using the strategy **dont care choose**(`extractrule`)*, only one of the two possible results is returned, e.g. one gets either* `a` *or* `b` *depending of the implementation of the strategies and of the AC-matcher.*

When using the strategy **dont know choose**(`extractrule`)*, the result of the application of the strategy is the set of two terms* `a` *and* `b`.

3.5.3 Handling several rules

We have just shown how non-determinism is handled when applying one rule. This is extended to the application of several rules, since it is natural to choose among several rules how to apply them, in a don't care or don't know way.

The application of the **dont know choose** strategy results in the application of all sub-strategies and yields the union of all results.

The application of the **dont care choose** strategy, returns the collection of results of one of the sub-strategy whose result is non-empty. If all sub-strategies fail, then it fails too, i.e. it yields the empty set.

The application of the **first** strategy, returns the collection of results of *the first* sub-strategy whose result is non-empty. If all sub-strategies fail, then it fails too, i.e. it yields the empty set.
Remark: We previously mentioned that the same rule label can be used for naming (different) rules. In fact this is a convenient way to express a strategy using different labels and a choice operator to link them.

Example 3.13 *Consider the following module:*

```
module testStrat                                                        1
                                                                        2
sort foo;                                                               3
 end                                                                    4
operators                                                               5
 global                                                                 6
 a :  foo;                                                              7
 b :  foo;                                                              8
 c :  foo;                                                              9
 d :  foo;                                                              10
 e :  foo;                                                              11
 f :  foo;                                                              12
 end                                                                    13
rules for foo                                                           14
 global                                                                 15
 [r1] a              => c      end                                      16
 [r2] a              => d      end                                      17
 [r3] b              => e      end                                      18
 [r4] b              => f      end                                      19
 [r5] c              => b      end                                      20
 end                                                                    21
strategies for foo                                                      22
 global                                                                 23
 [strat]                                                                24
 dont care choose(                                                      25
          dont know choose (r1)                                         26
          ||                                                            27
          dont know choose (r2)                                         28
          ||                                                            29
          dont know choose (r3 || r4))                                  30
 end                                                                    31
end                                                                     32
end                                                                     33
```

Then the strategy `strat` *applied to* `a` *results in* `c`*, and the same strategy applied to* `b` *results in* {`e`,`f`}.

3.5.4 Strategies concatenation

Two strategies are concatened by applying the second strategy on the all results of the first one.

Example 3.14 *In the same context as in the previous example (Example 3.13), the strategy:*

```
strategies for foo                                                      1
 global                                                                 2
 [conc]                                                                 3
          dont care choose (r1) ;                                       4
          dont care choose (r5) ;                                       5
          dont know choose (r3 ; r4)                                    6
 end                                                                    7
end                                                                     8
```

when applied on the term `a` *results in the application of first* `r1` *then* `r5` *and finally (dont know choose* `r3` *and* `r4`*), yielding the results* {`e`,`f`}.

3.5.5 Strategy iterators

In order to allow the automatic concatenation of the same strategy, ELAN offers three powerful iterators:

<iterator> ::= **iterate** (*<strategy>*)
| **repeat*** (*<strategy>*)
| **repeat+** (*<strategy>*)

The strategy **iterate** corresponds to applying zero, then one, then two, ... n times the strategy to the starting term, until the strategy fails. Thus (**iterate**(`s`))`t` returns $\bigcup_{n=0}^{\infty}(\mathtt{s}^n)\mathtt{t}$. Notice that **iterate** returns the results one by one even when an infinite derivation exists.

The strategy **repeat** iterates the strategy until it fails and return just the terms resulting of the last unfailing call of the strategy. The two variants are thus defined as:
(**repeat***(`s`))`t` = $(\mathtt{s}^n)\mathtt{t}$ where $(\mathtt{s}^{n+1})\mathtt{t}$ fails and $\mathtt{n} \geq 0$,
(**repeat+**(`s`))`t` = $(\mathtt{s}^n)\mathtt{t}$ where $(\mathtt{s}^{n+1})\mathtt{t}$ fails and $\mathtt{n} > 0$.

Example 3.15 *To illustrate the basic behaviour of these strategies, let us consider the following module:*

```
module stratFail                                                    1
import global int bool;                                             2
 end                                                                3
                                                                    4
sort foo ; end                                                      5
                                                                    6
operators global                                                    7
 a :                foo ;                                           8
 b :                foo ;                                           9
 end                                                               10
                                                                   11
rules for foo                                                      12
global                                                             13
 [a2b] a            => b       end                                14
 end                                                               15
                                                                   16
strategies for foo                                                 17
global                                                             18
 [try] dc(a2b) end                                                 19
 [repeatS] repeat*(dc(a2b)) end                                    20
 [repeatP] repeat+(dc(a2b)) end                                    21
 end                                                               22
end                                                                23
```

When applying the strategies on the terms `a` *or* `b`*, we are obtaining the following results:*

strategy	a	b
try	{b}	∅
repeatS	{b}	{b}
repeatP	{b}	∅

In particular, the difference between the **repeat*** *and* **repeat+** *is due to to fact that when* `dc(a2b)` *is applied zero time, this is by definition the identity strategy, returning the initial term.*

Example 3.16 *The extraction of the elements constituent of a list can be realized in the following way:*

```
module iterRepeat                                                          1
import global int bool;                                                    2
 end                                                                       3
                                                                           4
sort elem nelist list; end                                                 5
                                                                           6
operators global                                                           7
 @ :                ( int ) nelist ;                                       8
 @ .  @ :           ( int nelist ) nelist ;                                9
 element(@) :       ( nelist ) int ;                                       10
 end                                                                       11
                                                                           12
rules for int                                                              13
          e :  int;                                                        14
          l :  nelist;                                                     15
global                                                                     16
 [extract] element(e)                  => e       end                     17
 [extract] element(e.l)                => e       end                     18
 [extract] element(e.l)                => element(l) end                  19
 end                                                                       20
                                                                           21
strategies for int                                                         22
global                                                                     23
 [all0] dk(extract) end                                                    24
 [allRepS] repeat* ( dk(extract) ) end                                     25
 [allRepP] repeat+ ( dk(extract) ) end                                     26
 [allIter] iterate ( dk(extract) ) end                                     27
 end                                                                       28
                                                                           29
end                                                                        30
                                                                           31
```

We then obtain the following results:

strategy	1	element(1.2.3)
all0	∅	{1, element(2.3)}
allRepS	1	{1, 2, 3}
allRepP	∅	{1, 2, 3}
allIter	1	{element(1.2.3), element(2.3), element(3), 1, 2, 3}

3.5.6 The normalize strategy

Since labelled rules are directly applied only at the top of the terms, is is useful to have at hand a way to apply a family of labelled rules everywhere in a term to get it normalized. This is provided in ELAN by the **normalize** strategy. This strategy takes a strategy and use it in order to normalize the term on which it is applied.

▷ The is no assumption of the way the rules are applied, e.g. neither innermost nor outermost.

Example 3.17 *A typical example of the use of* **normalize** *is the normalization process of the typed lambda calculus, whose main module is described as follows:*

```
module normLambda                                                          1
import global int lterm;                                                   2
 local bool eq[lterm];                                                     3
 end                                                                       4
operators global                                                           5
```

```
 @ :  (int) lterm;                                                    6
 (@ @) :  (lterm lterm) lterm;                                        7
 la @ :  (lterm) lterm;                                               8
 end                                                                  9
rules for lterm                                                       10
          v, w       :  int;                                         11
          M, N, Q    :  lterm;                                       12
local                                                                 13
 [beta](la M N) => repl(1,M,N)                    end                 14
 [eta] (la M 1) => M if not free(1,M)             end                 15
 end                                                                  16
strategies for lterm                                                  17
global                                                                18
 [ss] normalise(dc(beta||eta))                                        19
 end                                                                  20
end                                                                   21
```

3.6 Evaluation mechanism

This section does not introduce any new syntactic material, but gives informal explanations on the operational semantics of the language. For a full description of the operational model, the reader in invited to consult [**?**].

As we have seen, there are two kinds of rules: labeled and unlabeled. They define two classes of rules that are applied on the term to be normalized in the following informal way:

– **Step 1** The current term is normalized using the unlabeled rules. This is done in order to perform functional evaluation and thus it is recommended to the user to provide a confluent and terminating unlabeled rewrite system to ensure termination and unicity of the result. This normalization process is built-in in the evaluation mechanism and consists in a leftmost innermost normalization. This should yield always a single result.

– **Step 2** Then one tries to apply on the normalized term a labeled rule following the strategy described in the logic description. This leads to a (possibly empty) collection of terms. If this set is empty, then the evaluation backtracks to the last choice point; if it is not empty, then the evaluation goes on by setting a new choice point and evaluating one of the returned terms by going to **step 1**.

In a slightly more formal way, a rule

$$\ell : l \to d \quad s_1, \ldots, s_n$$

where the s_i are either **where** or **if** expressions, is applied on a term t by:

1. Matching l against t. This computes a multiset of substitutions (because of equational matching). If this set contains more than two elements, one is chosen and the other ones are stored for possible future backtracking. Let σ be the chosen substitution.

2. The evaluation goes on by evaluating the expressions $s_1, \ldots, s_n$, one by one and in this *order* (i.e. from 1 to n).

3. If s_i is of the form **where** $x_i := (strat_i)t_i$, then one of the results (call it t'_i) of the application of the strategy $strat_i$ on the term t_i is chosen, and the substitution σ is extended by $x_i \mapsto t'_i$. The other results are stored for possible backtracking, and the evaluation goes on with s_{i+1}. If the strategy $strat_i$ fails on t_i, then we backtrack to the previous choice point.

4. If s_i is of the form **if** c_i, then the term c_i is evaluated following the normalization strategy. If the result is the `bool` constant `true`, then one evaluates the next expression s_{i+1}, otherwise one backtracks to s_{i-1}.

— Release Note: One important difference between version 1.17 and version V2.00 is that the order of evaluation of the **where** and **if** has been changed to be only *forward*, as described above.

Example 3.18 *This example shows an example of a brute force specification of the eight queens problem: How can one put eight queens on one chessboard in such a way that they do not interfere? In this example p1 refers to the position of the queen in the first column, p2 to the position of the second queen which should be in the second column and so on up to p8.*

```
module queens                                                          1
import                                                                 2
 bool int list[int];                                                   3
end                                                                    4
operators                                                              5
 global                                                                6
 queens              :  list[int];                                     7
 local                                                                 8
 ok(@,@,@)           :  ( int int list[int] ) bool;                    9
end                                                                    10
                                                                       11
rules for list[int]                                                    12
          p1,p2,p3,p4,p5,p6,p7,p8 :  int;                              13
          pp1,pp2,pp3,pp4,pp5,pp6,pp7,pp8 :  list[int];                14
local                                                                  15
 [queensrule] queens            => p8.pp7                              16
                                 -- first: the position on the first raw 17
                                 is choosen                            18
                                where p1:=(try) 0                      19
                                 -- second: the position on the second raw 20
                                 is also choosen                       21
                                where p2:=(try) 0                      22
                                where pp1:=() p1.nil                   23
                                 -- third: the positions on the first and 24
                                 second raws are checked to be compatible 25
                                if ok(1,p2,pp1)                        26
                                 -- quarto: choose a position on the third raw 27
                                 and continue ...                      28
                                where p3:=(try) 0                      29
                                where pp2:=() p2.pp1                   30
                                if ok(1,p3,pp2)                        31
                                where p4:=(try) 0                      32
                                where pp3:=() p3.pp2                   33
                                if ok(1,p4,pp3)                        34
                                where p5:=(try) 0                      35
                                where pp4:=() p4.pp3                   36
                                if ok(1,p5,pp4)                        37
                                where p6:=(try) 0                      38
                                where pp5:=() p5.pp4                   39
                                if ok(1,p6,pp5)                        40
                                where p7:=(try) 0                      41
                                where     pp6:=() p6.pp5               42
                                if ok(1,p7,pp6)                        43
                                where p8:=(try) 0                      44
                                where pp7:=() p7.pp6                   45
                                if ok(1,p8,pp7)                        46
 end                                                                   47
end                                                                    48
                                                                       49
rules for bool                                                         50
 p,d,diff : int;                                                       51
          l : list[int];                                               52
local                                                                  53
```

```
 [] ok(diff,d,nil)                  => true                                  end
 [] ok(diff,d,p.l)                  => false                  if d == p      end
 [] ok(diff,d,p.l)                  => false if d-p == diff end
 [] ok(diff,d,p.l)                  => false if p-d == diff end
 [] ok(diff,d,p.l)                  => ok(diff+1,d,l)                        end
end

rules for int
x:int;
local
 [tryrule] x              => 1      end
 [tryrule] x              => 2      end
 [tryrule] x              => 3      end
 [tryrule] x              => 4      end
 [tryrule] x              => 5      end
 [tryrule] x              => 6      end
 [tryrule] x              => 7      end
 [tryrule] x              => 8      end
end


strategies for int
local
 [try] dk(tryrule) end
end

strategies for list[int]
global
 [queens] dc(queensrule) end
end

end
```

Note that the strategy above returns only one solution. If it is changed to a **dont know choose**, *we get all possibilities.*

Note also the way all the numbers between 1 and 8 are generated in ELAN *using a know know choose strategy (see* `tryrule`*).*

▷ One should note that the term to be normalised is *first* reduced by ELAN using the non-labelled rules and the resulting term is then normalized using the given strategy. The following example illustrate this behavior

Example 3.19

```
module normalizeFirst

            The module identity is part of the standard library (see index)
import
          local identity[term] ;
 end

sort term ; end

operators
          global
          a           : term;
          f(@)        : (term) term;
end

rules for term
          x,y,z : term;
global
 [r1] f(x) => x end
```

```
 [] f(x) => x end                                                    20
end                                                                  21
                                                                     22
strategies for term                                                  23
global                                                               24
 [s1] dc(r1) end                                                     25
 [s2] dc( dc(r1) || dc(identity)) end                                26
 end                                                                 27
end                                                                  28
```

Applying the strategy `s1` *on the term* `f(a)` *gives no result since:*

1. `f(a)` *is normalized by the non-labelled rule into the term* `a`,
2. *then the normalisation process tries to apply the labelled rule* `r1` *on* `a` *and fails. So* no *strategy applies! The set of result in empty.*

When applying the strategy `s2` *on* `f(a)`*, for the same reason the reduction process fails to apply the rule* `r1` *but the strategy* `identity` *can then be applied and the result in then* `a` *since the term is first normalised.*

3.7 Modularity

The modularity constructions in the current version of ELAN are as simple as possible and their semantics is based on textual expansion.

The top-level information that should be provided by the super-user is the description of the logic (s)he wants to make available. This is achieved in ELAN by using two kinds of modules. The first one describes the top level of the logic and are presented in files having the '.lgi' extension, the second ones contain all subsequently needed informations. They are called element modules and they are presented in files having the extension '.eln'.

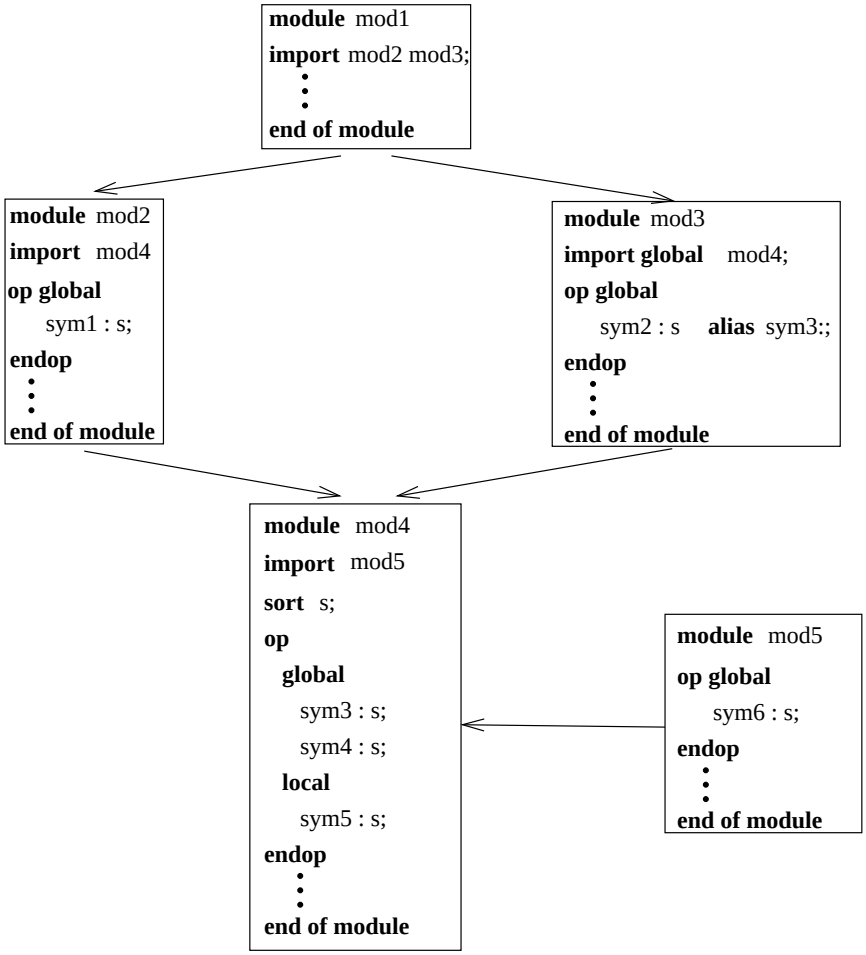
3.7.1 Visibility rules

The visibility rules in ELAN are the following:

- Only sorts are global. This means that they can be used in any module without any importation command. As a consequence, their name is unique in the ELAN universe built for a run.
- Operator, rewrite rules and strategies can on the contrary be local or global. A local operator will be known only in its own module (i.e. the module where it is defined). If an operator is declared to be global, then it is known in all modules importing the module where it is declared. This can be slightly modified, when qualifying the importation command by local or global.
- A special case is for aliases. The visibility rules for them are the same as for operators. The only interesting case is for a module that localy imports a signature but exports the alias definition. In this case, only the alias name of the operator will be known outside and not its original name: this is quite useful for renaming purposes.

— Release Note: The main change concerning the syntax of module is that now only sort are always global. See section 3.10 for a summary of all syntax modifications.

Example 3.20 *Let us consider the following importation graph:*



In this situation:

- *The sort* `s` *is visible in all modules,*
- `sym5` *is visible only in* `mod4`*,*
- `sym4` *is visible in modules* `mod1`*,* `mod2`*,* `mod3`*, and* `mod4`*,*
- `sym3` *is visible in the modules* `mod2`*,* `mod3` *and* `mod4` *and it is also visible in* `mod1` *but under the name* `sym2`*,*
- `sym2` *is just another name for* `sym3` *and it is only visible in module* `mod1` *and* `mod3`*,*
- `sym1` *is only visible in modules* `mod1` *and* `mod2`*.*

3.7.2 Built-in modules

For convenience and efficiency the following modules are built in ELAN:

1. `anyInteger` provides the sort `int` and all constants 0, 1, -1, 2, ... A description of this module is given in the standard library, see page 60.
2. `anyIdentifier` provides the sort `ident` and all constants a, b, c, aa, ab, ... A description of this module is given in the standard library, see page 60.

3. `anyQuotedIdentifier` provides all quoted constants "a", "b", "c", "aa", "ab", ... A description of this module is given in the standard library, see page 60.

These modules can be imported as standard ones.

3.7.3 Parameterized modules

Parameterized modules can be defined in ELAN as follows:

<formal module name> ::= *<identifier>*
| *<identifier>* [*<identifier>* { , *<identifier>* } *]

The module instanciation by its actual parameters is currently done by the syntactical replacement of the formal arguments by the actual ones.

Example 3.21 *The classical example is the one of parameterized lists:*

```
module simpleList[X]                                                          1
 sort X simpleList[X]; end                                                    2
 operators                                                                    3
          global                                                              4
          nil                 :                                simpleList[X]; 5
          @ .  @              :  ( X simpleList[X] )           simpleList[X]; 6
          append(@,@)         :  ( simpleList[X] simpleList[X] ) simpleList[X]; 7
 end                                                                          8
                                                                              9
 rules for simpleList[X]                                                      10
          e          :  X;                                                    11
          l,l1,l2    :  simpleList[X];                                        12
          global                                                              13
          [] append(nil, l)              => l                      end        14
          [] append(e.l1, l2)            => e .  append(l1,l2)     end        15
 end                                                                          16
end                                                                           17
```

and we get list of integers by instanciating `X` *by* `int` *and importing* `int` *as in:*

```
LPL simpleList description                                                    1
          query of sort       simpleList[int]                                 2
          result of sort      simpleList[int]                                 3
          import              int simpleList[int]                             4
          start with          ()query                                         5
end                                                                           6
```

Example 3.22 *Another classical example is based on the parameterized programming paradigm as advocated in [Gog90]:*

```
module map[X,Mod,Fun]                                                         1
import list[X] Mod; end                                                       2
 operators                                                                    3
          global                                                              4
          map_Fun(@):         ( list[X] )          list[X];                   5
 end                                                                          6
rules for list[X]                                                             7
          e :  X;                                                             8
 l :  list[X];                                                                9
          global                                                              10
          [] map_Fun(nil) => nil                              end              11
          [] map_Fun(e.l) => Fun(e) .  map_Fun(l)             end              12
 end                                                                          13
end                                                                           14
```

3.7.4 LGI modules

The top level description of the logic consists of:

1. the description of the syntax that the user is allowed to use in order to give his specifications.
2. the description of a term, called the query, that will be reduced in the rewrite logic defined by the (super) user.

The syntax is the following:

<lpl description> ::= **LPL** *<language name>* **description**
specification description
{ *<specification part description>* } $^+$
end
query
of sort *<query sort name>*
result of sort *<result sort name>*
import { *<module name>* } $^+$
[**check with** *<boolean term>*]
start with *<startterm>*
end

<specification part description> ::= **part** *<partname>*
of sort *<part sort name>*
import { *<module name>* } $^+$
[**check with** *<boolean term>*]

<specification> ::= **specification** *<specification name>*
{ *<partname>* *<term>* } $^+$
end

<language name> ::= *<identifier>*

<specification name> ::= *<identifier>*

<partname> ::= *<identifier>*

<part sort name> ::= *<sort name>*

<query sort name> ::= *<sort name>*

<result sort name> ::= *<sort name>*

<startterm> ::= < <result sort name> *term>*

Example 3.23 *If no specification is needed, the specification description part is skipped, as in the following logic description used for running Example 3.13:*

```
LPL testStrat description
```

```
query                of sort foo                                    3
 result              of sort foo                                    4
 import testStrat                                                   5
 start with (strat) query                                           6
                                                                    7
end                                                                 8
```

Example 3.24 *Here is a simple example of the top level description of how unification can be implemented:*

```
LPL unification description                                         1
          This part describes the syntax in which the user should   2
          complete the definition of the logic: i.e. how (s)he gives 3
          the specification.                                        4
specification description                                           5
                                                                    6
part Vars            of sort list[identifier]                       7
                     import sigSyntax                               8
part Ops             of sort list[pair[identifier,int]]             9
                     import sigSyntax                               10
end                                                                 11
          The following part describes the way the user should gives the query 12
          and how it will be evaluated                              13
query                of sort constraint                             14
                     result of sort constraint                      15
                     import unification[Vars,Ops]                   16
                                 constraint[Ops,Vars]               17
                                 termCommons[Ops,Vars]              18
                     start with (unify) query                       19
end                                                                 20
                                                                    21
```

In this unification logic, the user should provide the specification information as follows:

```
specification simplesig                                             1
 Vars x y z u v w                                                   2
 Ops a:0 b:0 c:0 f:2 g:1                                            3
end of specification                                                4
```

The user should use the (super-user defined) keywords `Vars` *and* `Ops` *to define respectively the variables and the operators, namely* `a` *of arity 0,* `f` *of arity 2, etc...*

Semantics

When parsing a specification, ELAN generates new modules that complete the definition of the rewrite theory to be used during the evaluation of the query. For instance, in the case of the previous example, these new modules are:

```
module Vars                                                         1
import anyidentifier signature; end                                 2
operators                                                           3
          global                                                    4
           Vars :  list[identifier];                                5
 end                                                                6
                                                                    7
rules for list[identifier]                                          8
          global                                                    9
           [] Vars => x y z ;                                       10
 end                                                                11
end                                                                 12
```

and

```
module Ops                                                          1
import anyidentifier signature; end                                 2
operators                                                           3
          global                                                    4
          Ops :  list[pair[identifier,number]];                     5
 end                                                                6
rules for list[pair[identifier,number]]                             7
          global                                                    8
          [] Ops => a:0 b:0 c:0 f:2 g:1 ;                           9
 end                                                                10
end                                                                 11
```

When a user provides a specification to an ELAN logic, this simply provides more information about the context in which the user wants to work, like the name of function symbols or basic axioms to use.

3.8 Pre-processing

Another original feature proposed by ELAN is the use of a pre-processing phase that allows to describe the logic to be encoded in an easier way.

As described in Figure 1.1, the pre-processor is performing textual replacements starting from informations given by the super-user in the element modules and the user in the specification file. The pre-processing phase in ELAN can just be thought of as a macro expansion mechanism which extends the parameterization process described before.

In order to express the syntax of the pre-processor construction, we need the notion of *constant expression* and of *text* defined as follows:

<constant expression> ::= *<number>*
| (*<subexpression>*)

<subexpression> ::= *<number>*
| *<subexpression>* + *<subexpression>*
| *<subexpression>* - *<subexpression>*
| *<subexpression>* * *<subexpression>*
| *<subexpression>* / *<subexpression>*
| *<subexpression>* % *<subexpression>*
| (*<subexpression>*)

<lexem> ::= *<identifier>*
| *<number>*
| *<char>*

<text> ::= { *<lexem>* } +

3.8.1 Simple duplication

A first feature is to allow simple duplications of a part of text. The syntax is:

<simple repetition> ::= *<lexem>* ~ *<constant expression>*
| { *<text>* } ~ *<constant expression>*

Example 3.25 *The text "*`a{,b}~3`*" is processed into "*`a, b, b, b`*".*

3.8.2 Duplication with argument

It is often necessary to allow description of objects like $f(t_1, \ldots, t_5)$. This is possible using the syntax:

<arg.repetition> ::= **{** *<text>* **}_** *<macro_name>* **=** *<first value>* **...** *<last value>*

<macro_name> ::= *<identifier>*

<first value> ::= *<constant expression>*

<last value> ::= *<constant expression>*

Example 3.26 *The text* "`P { & s_I=t_I }_I=1...3`" *is processed into* "`P & s_1=t_1 & s_2=t_2 & s_3=t_3`".

A special form of this duplication with arguments is the explicit construction of a list of indexed identifiers allowed using the syntax:

<arg.repetition> ::= *<identifier>* **_** *<number>* *<char>* **...** *<char>* *<identifier>* **_** *<number>*

Example 3.27 *The text* "`t_1,...,t_5`" *is processed into* "`t_1 , t_2 , t_3 , t_4 , t_5`".

3.8.3 Enumeration using FOR EACH

This construction allows to make the link between the specification given by the user and the logic described by the super-user. A natural motivation for this construction is given by the "standard" way inference rules are used. For example when describing how unification works, the following transformation rule is given:

Decompose $P \wedge f(s_1, \ldots, s_n) =^? f(t_1, \ldots, t_n) \rightarrow P \wedge s_1 =^? t_1 \wedge \ldots \wedge s_n =^? t_n$

It is generic in the sense that the operator f is unspecified. It can be a + of arity 2, or a *if@then@else@* of arity 3, or just a constant. We do not want, when specifying how the logic works, to give only the specific cases, we need to be as generic as possible.

ELAN provides via the FOR EACH construction of the pre-processor, a way to be generic. The syntax is the following:

<for each const.> ::= **FOR EACH** *<vardecl>* **SUCH THAT** *<varaffect>* **: {**
<text>
}

<vardecl> ::= *<varname>* { **,** *<varname>* }* **:** *<sort name>*
| *<vardecl>* **;** *<vardecl>*

<varaffect> ::= *<varname>* **:=** ([*<strategy name>*]) *<term>*
| *<varaffect>* **AND** *<varaffect>*
| *<varaffect>* **ANDIF** *<condition>*

<condition> ::= *<term>*

▷ Since the symbols { , } , ~ are reserved for pre-processor use, it is not possible to use them for any other purpose, even quoted.

▷ The pre-processor uses the character '_' (underline) for assembling identifiers. For example, the text 'a f _1_ x b" is pre-processed into the sequence of three identifiers a, f_1_x and b. The character '_' is thus part of an identifier even if there is a space between it and the rest of the string. It is thus better for the user not to use the _ symbol, except for building identifiers of course.

Example 3.28 *The rule* **Decomposition** *that we mentioned at the beginning of this section can be expressed in the following way:*

```
rules for eqSystem                                                                    1
          P : eqSystem;                                                               2
FOR EACH SS:pair[identifier,int]; F:identifier; N:int                                 3
SUCH THAT SS:=(listExtract) elem(Fss) AND F:=()first(SS)                              4
          AND N:=()second(SS) :{                                                      5
 rules for eqSystem                                                                   6
          s_1,...,s_N:term; t_1,...,t_N:term;                                         7
 local                                                                                8
 [decompose] P & F(s_1,...,s_N)=F(t_1,...,t_N)          => P { & s_I=t_I }_I=1...N end 9
 end                                                                                  10
}                                                                                     11
end                                                                                   12
```

If the specification given by the user consists in the operator symbols f *and* g *of respective arities 2 and 1, then the pre-processor expands the previous* **FOR EACH** *construction into:*

```
rules for eqSystem                                                                    1
          P : eqSystem;                                                               2
local                                                                                 3
 rules for eqSystem                                                                   4
          s_1,s_2 :  term;                                                            5
          t_1,t_2 :  term;                                                            6
 local                                                                                7
 [decompose] P & f(s_1,s_2)=f(t_1,t_2) => P & s_1=t_1 & s_2=t_2                       8
 end                                                                                  9
 end                                                                                  10
                                                                                      11
 rules for eqSystem                                                                   12
          s_1 :  term;                                                                13
          t_1 :  term;                                                                14
 local                                                                                15
 [decompose] P & g(s_1)=g(t_1) => P & s_1=t_1                                         16
 end                                                                                  17
 end                                                                                  18
end                                                                                   19
                                                                                      20
```

3.9 Interfacing Built-ins

ELAN allows to use external programs e.g. UNIF [**?**]. Here is how it works.

The external program is assumed to take as input a term and to return to ELAN a set of terms. One could imagine the input term as an input constraint and the returned terms as the solved forms computed by the program. The program is called as a sub-process of the ELAN session and it is linked to it via two UNIX pipes which are respectively attached to the standard input and output of the process (i.e. stdin stdout).

Since it is useful to reuse a given process as much as possible when allowed, the description of the called process gives the number of terms that can be send to the process before killing it and replacing it by another one.

Datas (i.e. terms) are send in and out under a textual form. This form is determined by the signature of the module in which the process call is performed.

The process call is realized through the use of the strategies **dccall** and **dkcall** (that stand for *dont care call* and *dont know call*), with three arguments.
– The first is the name of the process,
– the second is the number of terms that should be transmitted to the process before killing it,
– and last, the (unique) sort of the process results.
The syntax is thefore the following:

<strategy> ::= **dccall(** *<identifier>* **,** *<number>* **,** *<sort name>* **)**
| **dkcall(** *<identifier>* **,** *<number>* **,** *<sort name>* **)**

The difference between the two calls is that the first one returns all the results computed as the second one returns only the first result, even if several are computed.

In order to be able to communicate with ELAN, the process which is called should obey to the following syntatic restrictions.
The output should have the following syntax:

<processus output> ::= { # } + *<next solution>*

<next solution> ::= *<number>* { * } + *<term>* { # } + *<next solution>*
| **END** { # } +

where *<term>* is one result.
In the same way, the process should accept the syntax of the initial term, described as follows:

<processus input> ::= *<term>* *<next solution demand>*

<next solution demand> ::= **.** *<next solution demand>*
| **;**

where *<term>* is the term passed to the built-in process.

The way the communication is organised between ELAN and the built-in process is that ELAN is first sending the initial term to the process and every time it needs one of the computed term result, it sends the symbol "**.**" and wait for the answer of the process on its standard output. The answer should be in the syntax corresponding to the non-terminal *<next solution>* . If ELAN do not need anymore solution, it sends the symbol "**;**" which should provoke the process to compute the next result, or it sends an "end of file" if the limit of the process utilisation is reached.
▷ Since the synchronisation between ELAN and the called process in made through pipes, the two process should reinitialize their buffers after each large output. As a consequence the called process should call the fflush(stdout) procedure after returning each solution.

Example 3.29 *Assume that one would like to use the shell command "sort" for sorting a list of integers. The first step is to create a file containing the shell script calling the sort command (with the -n option since this is for numerical values) and that follows the syntactic conventions required by* ELAN *as just described above. Let us call this file "elansort". It contains the*

following informations:

*echo "## 0 **"*
sort -n
echo "END#"

Then the input-output syntax of integer lists should be adapted. Since the sort command could only be use for ordering the integers specified on each line of a file, we should write the file following this syntax.

The ELAN *module that realize this interface is as follow:*

```
module intListSort                                                          1
import global int list[int];                                                2
 operators local                                                            3
 @ '10' @            :  ( int list[int] ) list[int]          alias @.@:;    4
 end                                                                        5
import global int list[int]; end                                            6
                                                                            7
strategies for list[int]                                                    8
 [sortIntList]                                                              9
 dccall(elansort,1,list[int])                                               10
 end                                                                        11
end                                                                         12
```

Now applying the strategy `intListSort` *on a list of integers will result in the corresponding ordered list.*

3.10 Differences with previous version of the language

The language have evolved a lot between ELAN version 1.17 and ELAN V2.00. The changes concern the syntax of:

- modules
- rules, and
- strategies.

The main changes are emphasized in the corresponding section of the language description, and the main concern has been to uniformize the language constructions.

The transformations needed to change from the previous syntax to the current one are summarized below.

Concerning the declaration parts:

- `import ...` becomes `import ... end`
- `sort ...` becomes `sort ... end`
- `op ... endop` becomes `operators ... end`

Now, there is only one constructions to define rules, you have to replace:

- `rule for ...` by `rules for ...`
- `rule <name> for ...` by `rules for ... [name] ...`
- suppress `declare` keyword

- `body` or `bodies` by `global` or `local`
- textttend of rule, textttend of rules by `end`

Definition of strategies looks like rules definitions, you have to replace:

- `strategy ...` by `strategies for ... global|local [name] ...`
- `iterate ... enditerate` by `iterate(...)`
- `while ... endwhile` by `repeat*(...)`
- `repeat ... endrepeat` by `repeat*(...)`
- `dont know choose(r1 r2 r3)` by `dc(r1 || r2 || r3)`
- `repeat ... endrepeat dont care choose(...)` by `repeat*(...) ; dc(...)`
- `end of strategy` by `end`
- `end of module` by `end`

Local affectations and conditions are now evaluated from top to bottom. The order of presentation has been reversed, so that the *old* syntax:

```
[] l => r
        if c2
        where v2:=(...) ...
        where v1:=(...) ...
        if c1
```

becomes in the *new* syntax:

```
[] l => r
        if c1
        where v1:=(...) ...
        where v2:=(...) ...
        if c2
```

Chapter 4

ELAN: the system

This section describes the environment provided to use the ELAN language.

4.1 The parser

The ELAN parser is based on earley's one and is using sort informations in order to determine as soon as possible in the analyse the right choice. This explain why in the current version the language is requiring to give the sort information together with the term.

4.2 The interpreter

4.3 The compiler

Chapter 5

The standard library

This chapter presents the main features of the ELAN standard library. Remember that the path to this library is specified in your environment variable ELANLIB. ELAN can be used without any reference to this library, *except* for what concerns the use of the built-in objects.

This library has been designed to be small and as efficient as possible. In particular *no* AC operators is used. The resulting code is more efficient, at the price of sometimes heavier descriptions.

All this chapter is under revision. It is included in this version of the manual for keeping the related references properly.

5.1 The built-ins

You have the possibility in ELAN to start from nothing and to create your own world, using a non-conditional rewriting logic[1]. Nevertheless in most cases, users are interested in using standard data structures to build their own ones. So we provide standard useful built-ins.

5.1.1 Booleans

At the beginning there is nothing, so ELAN provides the true and false values and introduces the bool module. These two values are built-in and are deeply connected to the implementation of conditions in rewrite rules (c.f. Section 3.3.4).

`bool.eln`

To enrich the booleans, *polymorphic* equality, disequality and inequalities are defined and are also built-in:

`cmp.eln`

The next module is just restricting the operators exported by `cmp.eln`.

`eq.eln`

5.1.2 Numbers

Numbers can of course be created "by hand", but we choose in ELAN to provide built-in integers and floating point computations.

[1] Indeed rewriting with conditional rules is connected to the built-in booleans since firing a rule results from a positive match and the evaluation of the condition to the built-in value true.

First we provide the built-in module `anyInteger`:

`anyInteger.eln`

Then comes the module on integers:

`int.eln`

Floating point computations, as provided by the C compiler used for creating your ELAN version, are available using the `double` module.

`doubleConstants.eln`

`double.eln`

5.1.3 Identifiers

Two important built-in modules concern identifiers. First the standard ones (i.e. without quotes):

`anyIdentifier.eln`

and a similar version but with quotes:

`anyQuotedIdentifier.eln`

In fact quoted identifiers are often used when defining a logic in element modules in order to avoid syntactic conflicts at parsing time. Unquoted identifiers are mostly used in specifications. Then we introduce the module `ident` which provides the standard operations on identifiers:

`ident.eln`

and we recommand to use these operators via the module:

`identifier.eln`

5.1.4 Elementary term computations

Since it is of primarily use in symbolic computation on terms (and remember that everything in ELAN is a term except the built-ins), the occurrence relation and the replacement operation are provided built-ins, as described in the following two modules:

`occur.eln`

`replace.eln`

5.1.5 Summary of built-in codes

The following table summarises all the currently used codes. From the rank of the operator, one can infer the module in which it the operator is defined.

Code	Op	rank	comment
0	false	bool	
1	true	bool	
2			
3	+	(int int) int	
4	−	(int int) int	
5	*	(int int) int	
6	/	(int int) int	
7			
8	==	(int int) bool	
9	!=	(int int) bool	
10	<	(int int) bool	

continued on next page

continued from previous page

Code	Op	rank	comment
11	<=	(int int) bool	
12	>	(int int) bool	
13	>=	(int int) bool	
14	==	(ident ident) bool	
15	!=	(ident ident) bool	
16	replace	(X X) bool	
17	occur	(X X) bool	
18	equal	(X X) bool	
19	nequal	(X X) bool	
20	-	(int) int	
21	and	(bool bool) bool	
22	or	(bool bool) bool	
23	xor	(bool bool) bool	
24	not	(bool bool) bool	
25	b2i	(bool) int	
26	i2b	(int) bool	
27	%	(int int) int	modulo
28	and	(int int) int	bitwise and
29	or	(int int) int	bitwise or
30	less	(X X) bool	
31	lesseq	(X X) bool	
32	greater	(X X) bool	
33	greatereq	(X X) bool	
34	dconst1	(int) double	@.
35	dconst2	(int int) double	@.@
36	dconst3	(int int int) double	@.@E+@
37	dconst4	(int int int) double	@.@E-@
38	+	(double double) double	
39	-	(double double) double	
40	*	(double double) double	
41	/	(double double) double	
42	-	(double) double	
43	==	(double double) bool	
44	!=	(double double) bool	
45	<	(double double) bool	
46	<=	(double double) bool	
47	>	(double double) bool	
48	>=	(double double) bool	
49	doubleconstruct	(int int) double	
50	exp	(double) double	
51	exp2	(double) double	
52	exp10	(double) double	
53	log	(double) double	

continued on next page

continued from previous page

Code	Op	rank	comment
54	log2	(double) double	
55	log10	(double) double	
56	pow	(double) double	
57	sin	(double) double	
58	cos	(double) double	
59	tan	(double) double	
60	asin	(double) double	
61	acos	(double) double	
62	atan	(double) double	
63	i2double	(int) double	
64	dconst5	(int) double	.@
65	dconst6	(int int) double	.@E-@
66	dconst7	(int int) double	.@E-@
67	anyInt2int	−− > int	
68	anyIdent2ident	−− > ident	
71	dnconst1	(int) double	-@.
72	dnconst2	(int int) double	-@.@
73	dnconst3	(int int int) double	-@.@E+@
74	dnconst4	(int int int) double	-@.@E-@
75	dnconst5	(int) double	-.@
76	dnconst6	(int int) double	-.@E-@
77	dnconst7	(int int) double	-.@E-@
78	sqrt	(double)double	
79	−anyInt2int	−− > *int*	

Table 5.1: Built ins codes

5.2 Common ADT

A very easy module is the parameterized pairs, defined as follows:

`pair.eln`

The parameterized lists are defined, without surprise, in the standard way:

`list.eln`

and strategies are defined by:

`listCommons.eln`

5.3 Basic computations on terms

A good example of a parameterized module that uses its own reference is for term definition: in the current implementation, we first begin to define the common operations which do not depend on the concrete signature:

`term.eln`

Then we can define terms on a given signature. Note that one difficulty is that the signature is coming from the specification given by the user.

`termF.eln`

And finally we can define terms with variables:

`termV.eln`
`termFExt.eln`

Then one can specify how to apply a substitution on a term:

`applySubstOn.eln`

One can now define equations and system of equations:

`eqSystem.eln`

Syntactic unification can then be defined. Note that in this module, for efficiency reasons, no AC operator is used.

`syntacticUnification.eln`

5.4 Basic computations on atoms

This works like for terms.

`atom.eln`
`atomP.eln`
`atomPExt.eln`

5.5 Dealing with the user syntax

The next modules are describing a possible syntax for the user specifications. More complicated syntax (e.g. mixfix) can also be defined. The next module allows to parse signature:

`sigSyntax.eln`

For parsing Horn clauses, a possible syntax is given by:

`hornClauseSyntax.eln`

5.6 How to do nothing

It is sometimes useful to use an identity function or to use the reflexivity axiom of the rewrite logic. This is provided in the library by the following parametrised module.

`identity.eln`

Chapter 6

Contributed works

6.1 Description of medium size developments using ELAN

ELAN has been used in many different situations: let us mention some of them.

- The specification of the unification algorithm for higher-order unification based on explicit substitution [Bor95].
- The specification of constraint completion algorithms [KM95].
- The specification of constraint processings [Cas96].
- The combination of unification algorithms [Rin96].

ELAN also been used for specifying disunification, term orderings, a simple Constraint Logic Programming language, narrowing and constraint narrowing. Several examples are presented in [KKV95, Vit94].

6.2 ELAN's annotated bibliography

[KKV95] gives the first presentation of the general ideas developped in ELAN, with the definition of computational systems (including the definition of strategies) and the application of ELAN to the design and execution of constraint programming languages.

[Vit94] Describe the first main version of the ELAN system in full details, from design to implementation. In French (!) but we highly recommend to read it.

[BKK+96b] The general presentation of the ELAN system in its 1.17 version with a preview of the 2.0 features.

[Vit96] This describes the first ELAN compiler. A must is you are interested in compilation of underterministic normalisation. The first paper on compilation of strategies guided normalisation.

[BKK96a] A description of the operational as well as denotational semantics of the new ELAN strategies, available with version 2.0. This allows the user to define its own strategies as a computational system.

[KM96] Describes the reflective power of ELAN and of rewriting logic.

[KM95] Presents the implementation in ELAN of constraint completion.

[Bor95] describes the use of explicit substitutions for implementing in ELAN $\beta\eta$ normalisation as well as unification of higher-order unification.

[?] describes the use of ELAN to interface various unification algorithms and to manage the combination of them to solve complex problems.

[?] use an explicit substitution formalisation of the π-calculus for implementing input/output in ELAN.

[Cas96] describes the use of ELAN to specify various constraint manipulation algorithms like constraint local consistency and constraint propagation.

Acknowledgment

We are very grateful to Carlos Castro and Christophe Ringeissen for many fruitful discussions and constructive criticisms about ELAN for their careful reading of the manual and checking of examples. Many thanks also to Horatiu Cirstea and Jounaidi Benhassen for their feedbacks. Any remaining errors are of course ours.

Index

Bibliography

[BKK96a] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[BKK+96b] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1 & 2):203–216, April 1987.

[Bor95] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In Moroslav Bartošek, Jan Staudek, and Jiří Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer-Verlag, 1995.

[Cas96] Carlos Castro. Solving Binary CSP using Computational Systems. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[GKK+87] J. A. Goguen, Claude Kirchner, Hélène Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[Gog90] Joseph A. Goguen. Higher-order functions considered unnecessary for higher-order programming. In *Research Topics in Functional Programming*, The UT Year of Programming Series, chapter 12. Addison-Wesley Publishing Company, Inc., 1990.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalisme SDF - reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KM96] Hélène Kirchner and Pierre-Etienne Moreau. A reflective extension of Elan. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[Rin96] Ch. Ringeissen. Prototyping combination of unification algorithms with ELAN. Technical report, CRIN, 1996.

[Vit94] Marian Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[Vit96] Marian Vittek. A compiler for nondeterministic term rewriting systems. In Harald Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.